EXPERT'S VOICE® IN C++

SECOND EDITION

Exploring C++ 11

LEARN C++ WITH PRACTICAL HANDS-ON EXERCISES

Ray Lischner

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Contents at a Glance

Introduction

Hi, there. Thank you for reading my book *Exploring C++ 11*. My name is Ray, and I'll be your author today. And tomorrow. And the day after that. We'll be together for quite a while, so why don't you pull up a chair and get comfortable. My job is to help you learn C++. To do that, I have written a series of lessons called *Explorations*. Each Exploration is an interactive exercise that helps you learn C++ one step at a time. Your job is to complete the Explorations and, in so doing, learn C++.

No doubt you have already leafed through the book a little bit. If not, do so now. Notice that this book is different from most books. Most programming books are little more than written lectures. The author tells you stuff and expects you to read the stuff, learn it, and understand it.

This book is different. I don't see much point in lecturing at you. That's not how people learn best. You learn programming by reading, modifying, and writing programs. To that end, I've organized this book so that you spend as much time as possible reading, modifying, and writing programs.

How to Use This Book

Each exploration in this book is a mixture of text and interactive exercises. The exercises are unlike anything you've seen in other books. Instead of multiple-choice, fill-in-the-blank, or simple Q&A exercises, my lessons are interactive explorations of key C++ features. Early in the book, I will give you complete programs to work with. As you learn more C++, you will modify and extend programs. Pretty soon, you will write entire programs on your own.

By interactive, I mean that I ask questions and you answer them. I do my best to respond to your answers throughout the lesson text. It sounds crazy, but by answering the questions, you will be learning C++. To help ensure you answer the questions, I leave space in this book for you to write your answers. I'm giving you permission to write in this book (unless you are borrowing the book from a library or friend). In fact, I encourage you to write all your answers in the book. Only by answering the questions will you learn the material properly.

Sometimes, the questions have no right answer. I pose the question to make you ponder it, perhaps to look at a familiar topic from a new perspective. Other times, the question has an unambiguous, correct answer. I always give the answer in the subsequent text, so don't skip ahead! Write your answer before you continue reading. Then and only then can you check your answer. Some questions are tricky or require information that I have not yet presented. In such cases, I expect your answer to be wrong, but that's okay. Don't worry. I won't be grading you. (If you are using this book as part of a formal class, your teacher should grade this book's exercises solely on whether you complete them and never on whether your answer was correct. The teacher will have other exercises, quizzes, and tests to assess your progress in the class.) And no fair looking ahead and writing down the "correct" answer. You don't learn anything that way.

Ready? Let's practice.

What is your most important task when reading this book?

____

____

____

This question does not have a single correct answer, but it does have a number of demonstrably wrong answers. I hope you wrote something similar to, "Completing every exercise" or "Understanding all the material." Another good answer is, "Having fun."

The Book's Organization

C++ is a complicated language. To write even the most trivial program requires an understanding of many disparate aspects of the language. The language does not lend itself to neat compartmentalization into broad topics, such as functions, classes, statements, or expressions. This book, therefore, does not attempt such an organization. Instead, you learn C++ in small increments—a little bit of this, a little bit of that, some more of this—and pretty soon you will have accumulated enough knowledge to start writing nontrivial programs.

Roughly speaking, the book starts with basic expressions, declarations, and statements that are sufficient to work with simple programs. You learn how to use the standard library early in the book. Next, you learn to write your own functions, to write your own classes, to write your own templates, and then to write fairly sophisticated programs.

You won't be an expert, however, when you finish this book. You will need much more practice, more exposure to the breadth and depth of the language and library, and more practice. You will also need more practice. And some more. You get the idea.

Who Should Read This Book?

Read this book if you want to learn C++ and you already know at least one other programming language. You don't need to know a specific language or technology, however. In particular, you don't need to know C, nor do you need to know anything about object-oriented programming.

The C programming language influenced the design of many other languages, from PHP to Perl to AWK to C#, not to mention C++. As a result, many programmers who do not know C or C++ nonetheless find many language constructs hauntingly familiar. You might even feel confident enough to skip sections of this book that seem to cover old ground. Don't do that! From the start, the lessons present language features that are unique to C++. In a few, isolated cases, I will tell you when it is safe to skip a section, and only that section. Even when a language feature is familiar, it might have subtle issues that are unique to C++.

The trap is most perilous for C programmers because C++ bears the greatest superficial similarity with C. C programmers, therefore, have the most to overcome. By design, many C programs are also valid C++ programs, leading the unwary C programmer into the trap of thinking that good C programs are also good C++ programs. In fact, C and C++ are distinct languages, each with its own idioms and idiosyncrasies. To become an effective C++ programmer, you must learn the C++ way of programming. C programmers need to break some of their established habits and learn to avoid certain C features (such as arrays) in favor of better C++ idioms. The structure of this book helps you get started thinking in terms of C++, not C.

Projects

This book also contains four projects. The projects are opportunities to apply what you have learned. Each project is a realistic endeavor, based on the amount of C++ covered up to that point. I encourage you to try every project. Design your project using your favorite software design techniques. Remember to write test cases in addition to the source code. Do your best to make the code clean and readable, in addition to correct. After you are confident that your solution is finished, download the files from the book's web site, and compare your solution with mine.

Work Together

You can use this book alone, teaching yourself C++, or a teacher might adopt this book as a textbook for a formal course. You can also work with a partner. It's more fun to work with friends, and you'll learn more and faster by working together. Each of you needs your own copy of the book. Read the lessons and do the work on your own. If you have questions, discuss them with your partner, but answer the exercises on your own. Then compare answers with your partner. If your answers are different, discuss your reasoning. See if you can agree on a single answer before proceeding.

Work on the projects together. Maybe you can divide the work into two (or more) modules. Maybe one person codes and the other person checks. Maybe you'll practice some form of pair programming. Do whatever works best for you, but make sure you understand every line of code in the project. If you have asymmetric roles, be sure to swap roles for each project. Give everyone a chance to do everything.

For More Information

This book cannot teach you everything you need to know about C++. No single book can. After you finish this book, I encourage you to continue to read and write C++ programs and to seek out other sources of information. To help guide you, this book has a dedicated web site, `http://cpphelp.com/exploring/`. The web site has links to other books, other web sites, mailing lists, newsgroups, FAQs, compilers, other tools, and more. You can also download all the source code for this book, so you can save yourself some typing.

Why Explorations?

In case you were wondering about the unusual nature of this book, rest assured that, "though this be madness, yet there is method in't."

The method is an approach to teaching and writing that I developed while I was teaching computer science at Oregon State University. I wanted to improve the quality of my teaching, so I investigated research into learning and knowledge, especially scientific knowledge, and in particular, computer programming.

To summarize several decades of research: everyone constructs mental models of the world. We acquire knowledge by adding information to our models. The new information must always be in concert with the model. Sometimes, however, new information contradicts the model. In that case, we must adjust our models to accommodate the new information. Our brains are always at work, always taking in new information, always adjusting our mental models to fit.

As a result of this research, the emphasis in the classroom has shifted from teachers to students. In the past, teachers considered students to be empty vessels, waiting to be filled from the fount of the teacher's knowledge and wisdom. Students were passive recipients of information. Now we know better. Students are not passive, but active. Even when their outward appearance suggests otherwise, their brains are always at work, always absorbing new information and fitting that information into their mental models. The teacher's responsibility has changed from being the source of all wisdom to being an indirect manager of mental models. The teacher cannot manage those models directly but can only create classroom situations in which students have the opportunity to adjust their own models.

Although the research has focused on teachers, the same applies to authors. In other words, I cannot teach you C++, but I can create Explorations that enable you to learn C++. Explorations are not the only way to apply research to learning and writing, but they are a technique that I have refined over several years of teaching and have found successful. Explorations work because

- They force you to participate actively in the learning process. It's too easy to read a book passively. The questions force you to confront new ideas and to fit them into your mental model. If you skip the questions, you might also skip a crucial addition to your model.
- They are small, so your model grows in easy steps. If you try to grasp too much new information at once, you are likely to incorporate incorrect information into your model. The longer that misinformation festers, the harder it will be to correct. I want to make sure your model is as accurate as possible at all times.
- They build on what you know. I don't toss out new concepts with the vain hope that you will automatically grasp them. Instead, I tie new concepts to old ones. I do my best to ensure that every concept has a strong anchor in your existing mental model.
- They help you learn by doing. Instead of spending the better part of a chapter reading how someone else solves a problem, you spend as much time as possible working hands-on with a program: modifying existing programs and writing new programs.

C++ is a complicated language, and learning C++ is not easy. In any group of C++ programmers, even simple questions can often provoke varied responses. Most C++ programmers' mental models of the language are not merely incomplete but are flawed, sometimes in fundamental ways. My hope is that I can provide you with a solid foundation in C++, so that you can write interesting and correct programs, and most important, so that you can continue to learn and enjoy C++ for many years to come.

The C++ Standard

This book covers the current standard, namely, ISO/IEC 14882:2011 (E), Programming languages—C++. The 2011 edition of the standard is the all-new, improved standard, typically referred to as C++ 11. This book reflects new idioms, new language patterns, and new code. All the exercises have been tested on modern compilers. Most modern compilers do a decent job of conforming to the standard, although some fall down in key areas. The book's web site will have up-to-date details as vendors release updates to their compilers.

If you are stuck using C++ 03, I recommend using the first edition of this book. Although I occasionally mention which features are new to C++ 11, in general, I do not try to help C++ 03 users, and sometimes I completely avoid C++ 03 best practices, because they have been replaced by even better C++ 11 practices.

EXPLORATION 1

Honing Your Tools

Before you begin your exploration of the C++ landscape, you must gather some basic supplies: a text editor, a C++ compiler, a linker, and a debugger. You can acquire these tools separately or bundled, possibly as a package deal with an integrated development environment (IDE). Options abound, regardless of your platform, operating system, and budget.

If you are taking a class, the teacher will provide the tools or dictate which tools to use. If you are working at an organization that already uses C++, you probably want to use its tools, so you can become familiar with them and their proper use. If you have to acquire your own tools, check out this book's web site, `http://cpphelp.com/exploring/`. Tool versions and quality change too rapidly to provide details in print form, so you can find up-to-date suggestions on the web site. The following section gives some general advice.

Ray's Recommendations

C++ is one of the most widely used programming languages in the world; it is second only to C (depending on how you measure "widely used"). Therefore, C++ tools abound for many hardware and software environments and at a wide variety of price points.

You can choose command-line tools, which are especially popular in UNIX and UNIX-like environments, or you can opt for an IDE, which bundles all the tools into a single graphical user interface (GUI). Choose whichever style you find most comfortable. Your programs won't care what tools you use to edit, compile, and link them.

Microsoft Windows

If you are working with Microsoft Windows, I recommend Microsoft's Visual Studio (be sure that you have a current release). As I write this, the current release is Visual Studio 2012. If you want a no-cost option, download Visual Studio Express from Microsoft's web site (find a current link at `http://cpphelp.com`), or for an open-source solution, download MinGW, which is a port of the popular GNU compiler to Windows.

Note that C++/CLI is not the same as C++. It is a language that Microsoft invented to help integrate C++ into the .NET environment. That's why they chose a name that incorporates C++, just as the name C++ derives from the name C. Just as C++ is a distinct language from C, so is C++/CLI a distinct language from C++. This book covers standard C++ and nothing else. If you decide to use Visual Studio, take care that you work with C++, not C++/CLI or Managed C++ (the predecessor to C++/CLI).

Visual Studio includes a number of doodads, froufrous, and whatnots that are unimportant for your core task of learning C++. Perhaps you will have to use ATL, MFC, or .NET for your job, but for now, you can ignore all that. All you need is the C++ compiler and standard library.

If you prefer a free (as in speech) solution, the GNU compiler collection is available on Windows. Choose the Cygwin distribution, which includes a nearly complete UNIX-like environment, or MinGW, which is much smaller and might be easier to manage. In both cases, you get a good C++ compiler and library. This book's web site (`www.apress.com/9781430261933`) has links with helpful hints for installing and using these tools.

Macintosh OS X

Apple provides a no-cost IDE, call Xcode (see the link at http://cphelp.com), which includes the clang C++ compiler from the LLVM Compiler Infrastructure project. This is an excellent C++ compiler, and Apple bundles an excellent C++ standard library with it.

Everyone Else

I recommend the GNU compiler collection (GCC). The C++ compiler is called g++. Linux and BSD distributions typically come with the entire GCC suite, including g++, but you might have to install the necessary developer packages. Be sure you have a recent release.

Some hardware vendors (Oracle, HP, etc.) offer a commercial compiler specifically for their hardware. This compiler might offer better optimization than GCC but might not conform to the C++ standard as well as GCC. At least while you work through the exercises in this book, I recommend GCC. If you already have the vendor's compiler installed, and you don't want to bother installing yet another compiler, go ahead and use the vendor's compiler. However, if it ever trips over an example in this book, be prepared to install GCC.

If you are using an Intel hardware platform, Intel's compiler is excellent and available at no cost for noncommercial use. Visit the book's web site for a current link.

If you want to use an IDE, choose from Eclipse, NetBeans, KDevelop, Anjuta, and others (go to http://cpphelp.com for an up-to-date list).

Read the Documentation

Now that you have your tools, take some time to read the product documentation—especially the Getting Started section. Really, I mean it. Look for tutorials and other quick introductions that will help you get up to speed with your tools. If you are using an IDE, you especially need to know how to create simple command-line projects.

IDEs typically require you to create a project, workspace, or some other envelope, or wrapper, before you can actually write a C++ program. You must know how to do this, and I can't help you, because every IDE is different. If you have a choice of project templates, choose "console," "command-line," "terminal," "C++ Tool," or some project with a similar name.

How long did it take you to read the documentation for your compiler and other tools?

____

Was that too much time, too little time, or just right? _______________

The C++ language is subject to an international standard. Every compiler (more or less) adheres to that standard but also throws in some nonstandard extras. These extras can be useful—even necessary—for certain projects, but for this book, you have to make sure you use only standard C++. Most compilers can turn off their extensions. Even if you hadn't read the documentation before, do so now, to find out which options you need to enable you to compile standard C++ and only standard C++.

Write down the options, for future reference.

____

____

____

You may have missed some of the options; they can be obscure. To help you, Table 1-1 lists the command-line compiler options you need for Microsoft Visual C++, g++, and clang. This book's web site has suggestions for some other popular compilers. If you are using an IDE, look through the project options or properties to find the equivalents.

Table 1-1. *Compiler Options for Standard C++*

Compiler	Options
Visual C++ command line	`/EHsc /Za`
Visual C++ IDE	Enable C++ exceptions, disable language extensions
g++	`-pedantic -std=c++11`
clang/llvm	`-pedantic -std=c++11`

Your First Program

Now that you have your tools, it's time to begin. Fire up your favorite text editor or your C++ IDE and start your first project or create a new file. Name this file `list0101.cpp`, which is short for Listing 1-1. Several different file-name extensions are popular for C++ programs. I like to use `.cpp`, where the *p* means "plus." Other common extensions are `.cxx` and `.cc`. Some compilers recognize `.C` (uppercase *C*) as a C++ file extension, but I don't recommend using it, because it is too easy to confuse with `.c` (lowercase *c*), the default extension for C programs. Many desktop environments do not distinguish between uppercase and lowercase file names, further compounding the problem. Pick your favorite and stick with it. Type in the text contained within Listing 1-1. (With one exception, you can download all the code listings from this book's web site. Listing 1-1 is that exception. I want you to get used to typing C++ code in your text editor.)

Listing 1-1. Your First C++ Program

```
/// Sort the standard input alphabetically.
/// Read lines of text, sort them, and print the results to the standard output.
/// If the command line names a file, read from that file. Otherwise, read from
/// the standard input. The entire input is stored in memory, so don't try
/// this with input files that exceed available RAM.

#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

void read(std::istream& in, std::vector<std::string>& text)
{
   std::string line;
   while (std::getline(in, line))
       text.push_back(line);
}

int main(int argc, char* argv[])
{
   // Part 1. Read the entire input into text. If the command line names a file,
   // read that file. Otherwise, read the standard input.
   std::vector<std::string> text; ///< Store the lines of text here
   if (argc < 2)
     read(std::cin, text);
   else
   {
      std::ifstream in(argv[1]);
      if (not in)
      {
         std::perror(argv[1]);
         return EXIT_FAILURE;
      }
      read(in, text);
   }

   // Part 2. Sort the text.
   std::sort(text.begin(), text.end());

   // Part 3. Print the sorted text.
   std::copy(text.begin(), text.end(),
             std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

No doubt, some of this code is gibberish to you. That's okay. The point of this exercise is not to understand C++ but to make sure you can use your tools properly. The comments describe the program, which is a simple sort utility. I could have started with a trivial, "Hello, world"–type of program, but that touches only a tiny fraction of the language and library. This program, being slightly more complex, does a better job at revealing possible installation or other problems with your tools.

Now go back and double-check your source code. Make sure you entered everything correctly.

Did you actually double-check the program? ________________

Did you find any typos that needed correcting? ________________

To err is human, and there is no shame in typographical errors. We all make them. Go back and recheck your program.

Now compile your program. If you are using an IDE, find the Compile or Build button or menu item. If you are using command-line tools, be sure to link the program too. For historical (or hysterical) reasons, UNIX tools such as g++ typically produce an executable program named a.out. You should rename it to something more useful or use the -o option to name an output file. Table 1-2 shows sample command lines to use for Visual C++, g++, and clang.

Table 1-2. *Sample Command Lines to Compiler `list0101.cpp`*

Compiler	Command Line
Visual C++	`cl /EHsc /Za list0101.cpp`
g++	`g++ -o list0101 -pedantic -std=c++11 list0101.cpp`
Clang	`clang++ -o list0101 -pedantic -std=c++11 list0101.cpp`

If you receive any errors from the compiler, it means you made a mistake entering the source code; the compiler, linker, or C++ library has not been installed correctly; or the compiler, linker, or library does not conform to the C++ standard and so are unsuitable for use with this book. Triple-check that you entered the text correctly. If you are confident that the error lies with the tools and not with you, check the date of publication. If the tools predate 2011, discard them immediately. They predate the standard and, therefore, by definition, they cannot conform to the standard. Compiler vendors have worked hard at ensuring their tools conform to the latest standard, but this takes time. Only in 2013 have we seen compilers that truly implement enough of the C++ 11 standard to be useful.

If all else fails, try a different set of tools. Download the current release of GCC or Visual Studio Express. You may have to use these tools for this book, even if you must revert to some crusty, rusty, old tools for your job.

Successful compilation is one thing, but successful execution is another. How you invoke the program depends on the operating system. In a GUI environment, you will need a console or terminal window where you can type a command line. You may have to type the complete path to the executable file or only the program name—again, this depends on your operating system. When you run the program, it reads text from the standard input stream, which means whatever you type, the program reads. You then have to notify the program you are done, by pressing the magic keystrokes that signal end-of-file. On most UNIX-like operating systems, press Control+D. On Windows, press Control+Z.

Running a console application from an IDE is sometimes tricky. If you aren't careful, the IDE might close the program's window before you have a chance to see any of its output. You have to ensure that the window remains visible. Some IDEs (such as Visual Studio and KDevelop) do this for you automatically, asking you to press a final Enter key before it closes the window.

If the IDE doesn't keep the window open automatically, and you can't find any option or setting to keep the window open, you can force the issue by setting a break point on the program's closing curly brace or the nearest statement where the debugger will let you set a break point.

Knowing how to run the program is one thing; another is to know what numbers to type so you can test the program effectively. **How would you test list0101 to ensure it is running correctly?**

____

____

____

____

____

Okay, do it. **Does the program run correctly?** ________________

There, that was easy, wasn't it? You should try several different sequences. Run the program with no input at all. Try it with one line of text. Try it with two that are already in order and two lines that are in reverse order. Try a lot of text, in order. Try a lot of text in random order. Try text in reverse order.

Before you finish this Exploration, I have one more exercise. This time, the source file is more complicated. It was written by a professional stunt programmer. Do not attempt to read this program, even with adult supervision. Don't try to make any sense of the program. Above all, don't emulate the programming style used in this program. This exercise is not for you, but for your tools. Its purpose is to see whether your compiler can correctly compile this program and that your library implementation has the necessary parts of the standard library. It's not a severe torture test for a compiler, but it does touch on a few advanced C++ features.

So don't even bother trying to read the code. Just download the file `list0102.cpp` from the book's web site and try to compile and link it with your tools. (I include the full text of the program only for readers who lack convenient Internet access.) If your compiler cannot compile and run Listing 1-2 correctly, you must replace it (your compiler, not the program). You may be able to squeak by in the early lessons, but by the end of the book, you will be writing some fairly complicated programs, and you need a compiler that is up to the task. (By the way, Listing 1-2 pretty much does the same thing as Listing 1-1.)

Listing 1-2. Testing Your Compiler

```
/// Sort the standard input alphabetically.
/// Read lines of text, sort them, and print the results to the standard output.
/// If the command line names a file, read from that file. Otherwise, read from
/// the standard input. The entire input is stored in memory, so don't try
/// this with input files that exceed available RAM.
///
/// Comparison uses a locale named on the command line, or the default, unnamed
/// locale if no locale is named on the command line.

#include <algorithm>
#include <cstdlib>
#include <fstream>
#include <initializer_list>
#include <iostream>
#include <locale>
#include <string>
#include <vector>

template<class C>
struct text : std::basic_string<C>
{
  text() : text{""} {}
  text(char const* s) : std::basic_string<C>(s) {}
  text(text&&) = default;
  text(text const&) = default;
  text& operator=(text const&) = default;
  text& operator=(text&&) = default;
};

/// Read lines of text from @p in to @p iter. Lines are appended to @p iter.
/// @param in the input stream
/// @param iter an output iterator
template<class Ch>
auto read(std::basic_istream<Ch>& in) -> std::vector<text<Ch>>
{
    std::vector<text<Ch>> result;
    text<Ch> line;
    while (std::getline(in, line))
        result.push_back(line);

    return result;
}

/// Main program.
int main(int argc, char* argv[])
try
{
    // Throw an exception if an unrecoverable input error occurs, e.g.,
    // disk failure.
    std::cin.exceptions(std::ios_base::badbit);
    // Part 1. Read the entire input into text. If the command line names a file,
    // read that file. Otherwise, read the standard input.
    std::vector<text<char>> text; ///< Store the lines of text here
    if (argc < 2)
        text = read(std::cin);
    else
    {
        std::ifstream in(argv[1]);
        if (not in)
        {
            std::perror(argv[1]);
            return EXIT_FAILURE;
        }
        text = read(in);
    }

    // Part 2. Sort the text. The second command line argument, if present,
    // names a locale, to control the sort order. Without a command line
    // argument, use the default locale (which is obtained from the OS).
    std::locale const& loc{ std::locale(argc >= 3 ? argv[2] : "") };
    std::collate<char> const& collate( std::use_facet<std::collate<char>>(loc) );
    std::sort(text.begin(), text.end(),
        [&collate](std::string const& a, std::string const& b)
        {
            return collate.compare(a.data(), a.data()+a.size(),
                                   b.data(), b.data()+b.size()) < 0;
        }
    );

    // Part 3. Print the sorted text.
   for (auto const& line :  text)
      std::cout << line << '\n';
}
catch (std::exception& ex)
{
    std::cerr << "Caught exception: " << ex.what() << '\n';
    std::cerr << "Terminating program.\n";
    std::exit(EXIT_FAILURE);
}
catch (...)
{
    std::cerr << "Caught unknown exception type.\nTerminating program.\n";
    std::exit(EXIT_FAILURE);
}
```

I caught you peeking. In spite of my warning, you tried to read the source code, didn't you? Just remember that I deliberately wrote this program in a complicated fashion to test your tools. By the time you finish this book, you will be able to read and understand this program. Even more important, you will be able to write it more simply and more cleanly. Before you can run, however, you must learn to walk. Once you are comfortable working with your tools, it's time to start learning C++. The next Exploration begins your journey with a reading lesson.

EXPLORATION 2

Reading C++ Code

I suspect you already have some knowledge of C++. Maybe you already know C, Java, Perl, or other C-like languages. Maybe you know so many languages that you can readily identify common elements. Let's test my hypothesis. Take a few minutes to read Listing 2-1, then answer the questions that follow it.

Listing 2-1. Reading Test

```
/// Read the program and determine what the program does.

#include <iostream>
#include <limits>

int main()
{
    int min{std::numeric_limits<int>::max()};
    int max{std::numeric_limits<int>::min()};
    bool any{false};
    int x;
    while (std::cin >> x)
    {
        any = true;
        if (x < min)
            min = x;
        if (x > max)
            max = x;
    }

    if (any)
        std::cout << "min = " << min << "\nmax = " << max << '\n';
}
```

What does Listing 2-1 do?

Listing 2-1 reads integers from the standard input and keeps track of the largest and smallest values entered. After exhausting the input, it then prints those values. If the input contains no numbers, the program prints nothing.

Let's take a closer look at the various parts of the program.

Comments

Line 1 begins with three consecutive slashes to start a comment. The comment ends at the end of the line. Actually, you need only two slashes to signal the start of a comment (`//`), but as you will learn later in the book, the extra slash has a special meaning.

Note that you cannot put a space between the slashes. That's true in general for all the multicharacter symbols in C++. It's an important rule, and one you must internalize early. A corollary to the "no spaces in a symbol" rule is that when C++ sees adjacent characters, it usually constructs the longest possible symbol, even if you can see that doing so would produce meaningless results.

The other method you can use to write a comment in C++ is to begin the comment with `/*` and end it with `*/`. The difference between this style and the style demonstrated in Listing 2-1 is with this method, your comment can span multiple lines. You may notice that some programs in this book use `/**` to start a comment. Much like the third slash in Listing 2-1, this second asterisk (*) is magic, but unimportant at this time. A comment cannot nest within a comment of the same style, but you can nest one style of comment in comments of the other style, as illustrated in Listing 2-2.

Listing 2-2. Demonstrating Comment Styles and Nesting

```
/* Start of a comment /* start of comment characters are not special in a comment
 // still in a comment
 Still in a comment
*/
no_longer_in_a_comment();
// Start of a comment /* start of comment characters are not special in a comment
no_longer_in_a_comment();
```

The C++ community uses both styles widely. Get used to seeing and using both styles.

Modify Listing 2-1 to change the `///` comment to use the `/** ... */` style, then try to recompile the program. **What happens?**

____

If you made the change correctly, the program should still compile and run normally. The compiler eliminates comments entirely, so nothing about the final program should be different. (With one exception being that some binary formats include a time stamp, which would necessarily differ from one compilation run to another.)

Headers

Lines 3 and 4 of Listing 2-1 import declarations and definitions from parts of the standard library. C++, like C and many other languages, distinguishes between the core language and the standard library. Both are part of the standard language, and a tool suite is incomplete without both parts. The difference is that the core language is self-contained. For example, certain types are built-in, and the compiler inherently knows about them. Other types are defined in terms of the built-in types, so they are declared in the standard library, and you must instruct the compiler that you want to use them. That's what lines 3 and 4 are all about.

In particular, line 3 informs the compiler about the names of the standard I/O streams (`cin` for the standard input and `cout` for the standard output), the input operator (`>>`), and the output operator (`<<`). Line 4 brings in the name `numeric_limits`. Note that names from the standard library generally begin with `std::` (short for "standard").

In C++ parlance, the #include keyword is also a verb, as in “line 3 *includes* the iostream header,” “line 4 includes the limits header,” and so on. A *header* is typically a file that contains a series of declarations and definitions. (A declaration is a kind of definition. A definition tells the compiler more about a name than a declaration. Don’t worry about the difference yet, but notice when I use *declaration* and when I use *definition*.) The compiler needs these declarations and definitions, so it knows what to do with names such as std::cin. Somewhere in the documentation for your C++ compiler and standard library is the location for its standard headers. If you are curious, visit that folder or directory and see what you can find there, but don’t be disappointed if you can’t read the headers. The C++ standard library makes full use of the entire range of C++ language features. It’s likely you won’t be able to decipher most of the library until after you’ve made it through a large part of this book.

Another important C++ rule: the compiler has to know what every name means. A human can often infer meaning or at least a part of speech from context. For example, if I were to say, “I furbled my drink all over my shirt,” you may not know exactly what *furbled* means, but you can deduce that it is the past tense of a verb and that it probably implies something undesirable and somewhat messy.

C++ compilers are a lot dumber than you. When the compiler reads a symbol or identifier, it must know exactly what the symbol or identifier means and what part of “speech” it is. Is the symbol a punctuator (such as the statement-ending semicolon) or an operator (such as a plus sign for addition)? Is the identifier a type? A function? A variable? The compiler also has to know everything you can do with that symbol or name, so it can correctly compile the code. The only way it can know is for you to tell it, and the way you tell it is by writing a declaration or by importing a declaration from a header. And that’s what #include directives are all about.

Any line that begins with # ends at the end of the line. C++ has several different # directives, but only #include concerns us at this time. Inside the angle brackets must be a header name, which is typically a standard library header, but it might also be a header from a third-party library.

Later in the book, you’ll even learn to write your own headers.

Modify line 4 to misspell *limits* as *stimil*. Try to compile the program. **What happens?**

____

____

____

The compiler cannot find any header named stimil, so it issues a message. Then it may try to compile the program, but it doesn’t know what std::numeric_limits is, so it issues one or more messages. Some compilers cascade messages, which means every use of std::numeric_limits produces additional messages. The actual error becomes lost in the noise. Focus on the first one or few messages the compiler issues. Fix them, then try again. As you gain experience with C++, you will learn which messages are mere noise and which are important. Unfortunately, most compilers will not tell you, for example, that you can’t use std::numeric_limits until you include the <limits> header. Instead, you need a good C++ language reference, so you can look up the correct header on your own. The first place to check is the documentation that accompanies your compiler and library. Authors have been slower than compiler-writers to catch up to the C++ 11 standard, so keep checking the web site and bookstores for updated references.

Most programmers don’t use <limits> much; Listing 2-1 included it only to obtain the definition of std::numeric_limits. On the other hand, almost every program in this book uses <iostream>, because it declares the names and types of the I/O stream objects, std::cin and std::cout. There are other I/O headers, but for basic console interactions, you need only <iostream>. You will meet more headers in coming Explorations.

Main Program

Every C++ program must have int main(), as shown on line 6. You are permitted a few variations on a theme, but the name main is crucial. A program can have only one main, and the name must be spelled using all lowercase characters. The definition must start with int.

■ **Note** Some books instruct you to use `void`. Those books are wrong. If you have to convince someone that `void` is wrong and int is right, refer the skeptic to section 3.6.1 of the C++ Standard.

For now, use empty parentheses after the name `main`.

The next line starts the main program. Notice how the statements are grouped inside curly braces (`{` and `}`). That's how C++ groups statements. A common error of novices is to omit a curly brace or miss seeing them when reading a program. If you are used to more verbose languages, such as Pascal, Ada, or Visual Basic, you might need some time acquainting yourself with the more terse C++ syntax. This book will give you plenty of opportunities to practice.

Modify line 6 to spell `main` in capital letters (`MAIN`). Try to compile the program. **What happens?**

____

____

____

The compiler probably accepts the program, but the linker complains. Whether you can see the difference between the compiler and the linker depends on your particular tools. Nonetheless, you failed to create a valid program, because you must have a `main`. Only the name *main* is special. As far as the compiler is concerned, MAIN is just another name, like *min* or *max*. Thus, you don't get an error message saying that you misspelled *main*, only that *main* is missing. There's nothing wrong with having a program that has a function named MAIN, but to be a complete program, you must be sure to include the definition `main`.

Variable Definitions

Lines 8 through 11 define some variables. The first word on each line is the variable's type. The next word is the variable name. The name is followed optionally by an initial value in curly braces. The type `int` is short for *integer*, and `bool` is short for *Boolean*.

■ **Note** Boolean is named after George Boole, the inventor of mathematical logic. As such, some languages use the name logical for this type. It is unclear why languages such as C++ use `bool` instead of `boole` for the type named after Boole.

The name `std::numeric_limits` is part of the C++ standard library and lets you query the attributes of the built-in arithmetic types. You can determine the number of bits a type requires, the number of decimal digits, the minimum and maximum values, and more. Put the type that you are curious about in angle brackets. (You'll see this approach to using types quite often in C++.) Thus, you could also query `std::numeric_limits<bool>::min()` and get false as the result.

If you were to query the number of bits in bool, what would you expect as a result? ______________

Try compiling and running Listing 2-3, and find out if you are correct.

Listing 2-3. Determining the Number of Bits in a `bool`

```
#include <iostream>
#include <limits>
```

```
int main()
{
  // Note that "digits" means binary digits, i.e., bits.
  std::cout << "bits per bool: " << std::numeric_limits<bool>::digits << '\n';
}
```

Did you get the value you expected? If not, do you understand why you got 1 as a result?

Statements

Line 12 of Listing 2-1 contains a while statement. Lines 15, 17, and 21 begin if statements. They have similar syntax: both statements begin with a keyword, followed by a Boolean condition in parentheses, followed by a statement. The statement can be a simple statement, such as the assignment on line 16, or it can be a list of statements within curly braces. Notice that a simple statement ends with a semicolon.

Assignment (lines 14, 16, and 18) uses a single equal sign. For clarity, when I read a program out loud or to myself, I like to read the equal sign as "gets." For example, "x gets min."

A `while` loop performs its associated statement while the condition is true. The condition is tested prior to executing the statement, so if the condition is false the first time around, the statement never executes.

On line 12, the condition is an input operation. It reads an integer from the standard input (`std::cin`) and stores that integer in the variable `x`. The condition is true as long as a value is successfully stored in `x`. If the input is malformed, or if the program reaches the end of the input stream, the logical condition becomes false, and the loop terminates.

The `if` statement can be followed by an `else` branch; you'll see examples in future Explorations.

Line 21's condition consists of a single name: `any`. Because it has type `bool`, you can use it directly as a condition.

Modify line 15 to change the statement to just "`if (x)`". This kind of mistake sometimes occurs when you get careless (and we all get careless from time to time). **What do you expect to happen when you compile the program?**

____

____

Were you surprised that the compiler did not complain? **What do you expect to happen when you run the program?**

____

____

____

If you supply the following input to the program, what do you expect as output?

```
0   1   2   3
```

____

____

If you supply the following input to the program, what do you expect as output?

```
3   2   1   0
```

____

____

Explain what is happening:

____

____

____

____

____

C++ is permissive about what it allows as a condition. Any numerical type can be a condition, and the compiler treats non-zero values as true and zero as false. In other words, it supplies an implicit ≠ 0 to test the numeric value.

Many C and C++ programmers take advantage of the brevity these languages offer, but I find it a sloppy programming practice. Always make sure your conditions are logical in nature, even if that means using an explicit comparison to zero. The C++ syntax for comparing ≠ is `!=`, as in `x != 0`.

Output

The output operator is `<<`, which your program gets by including `<iostream>`. You can print a variable's value, a character string, a single character, or a computed expression.

Enclose a single character in single quotes, such as `'X'`. Of course, there may be times when you have to include a single quote in your output. To print a single quote, you will have to escape the quote character with a backslash (`\'`). Escaping a character instructs the compiler to process it as a standard character, not as a part of the program syntax. Other escape characters can follow a backslash, such as `\n` for a newline (that is, a magic character sequence to start a new line of text; the actual characters in the output depend on the host operating system). To print a backslash character, escape it: `'\\'`. Some examples of characters include the following: `'x'`, `'#'`, `'7'`, `'\\'`, `'\n'`.

If you want to print more than one character at a time, use a character string, which is enclosed in double quotes. To include a double quote in a string, use a backslash escape:

```
std::cout << "not quoted; \"in quotes\", not quoted";
```

A single output statement can use multiple occurrences of `<<`, as shown in line 22, or you can use multiple output statements. The only difference is readability.

Modify Listing 2-3 to experiment with different styles of output. Try using multiple output statements.

Remember to use curly braces when the body of an if statement contains more than one statement.

See! I told you that you could read a C++ program. Now all you have to do is fill in some of your knowledge gaps about the details. The next Exploration starts doing that with the basic arithmetic operators.

EXPLORATION 3

Integer Expressions

In Exploration 2, you examined a program that defined a few variables and performed some simple operations on them. This Exploration introduces the basic arithmetic operators. Read Listing 3-1, then answer the questions that follow it.

Listing 3-1. Integer Arithmetic

```
 1 /// Read the program and determine what the program does.
 2
 3 #include <iostream>
 4
 5 int main()
 6 {
 7     int sum{0};
 8     int count{};
 9     int x;
10     while (std::cin >> x)
11     {
12         sum = sum + x;
13         count = count + 1;
14     }
15
16     std::cout << "average = " << sum / count << '\n';
17 }
```

What does the program in Listing 3-1 do?

____

____

Test the program with the following input:

```
10    50    20    40    30
```

Lines 7 and 8 initialize the variables `sum` and `count` to zero. You can enter any integer value in the curly braces to initialize a variable (line 7); the value does not have to be constant. You can even leave the curly braces empty to initialize the variable to a suitable default value (e.g., `false` for `bool`, `0` for `int`), as shown on line 8. Without any curly braces, the variable is not initialized, so the only action the program can do is to assign a new value to the variable, as shown on lines 9 and 10. Ordinarily, it's a bad idea not to initialize your variables, but in this case, `x` is safe, because line 10 immediately stuffs a value into it by reading from the standard input.

Lines 12 and 13 show examples of addition (+) and assignment (=). Addition follows the normal rules of computer arithmetic (we'll worry about overflow later). Assignment works the way it does in any procedural language.

Thus, you can see that Listing 3-1 reads integers from the standard input, adds them up, and prints the average (mean) value, as computed by the division (/) operator. Or does it?

What is wrong with Listing 3-1?

____

____

____

Try running the program with no input—that is, press the end-of-file keystroke immediately after starting the program. Some operating systems have a "null" file that you can supply as the input stream. When a program reads from the null file, the input stream always sees an end-of-file condition. On UNIX-like operating systems, run the following command line:

```
list0301 < /dev/null
```

On Windows, the null file is called NUL, so type

```
list0301 < NUL
```

What happens?

____

C++ doesn't like division by zero, does it? Each platform reacts differently. Most systems indicate an error condition one way or another. A few quietly give you garbage results. Either way, you don't get anything meaningful.

Fix the program by introducing an if statement. Don't worry that the book hasn't covered if statements yet. I'm confident you can figure out how to ensure this program avoids dividing by zero. **Write the corrected program below**:

____

____

____

____

____

____

____

____

____

____

____

____

____

____

Now try your new program. **Was your fix successful?**

Compare your solution with Listing 3-2.

Listing 3-2. Print Average, Testing for a Zero Count

```
/// Read integers and print their average.
/// Print nothing if the input is empty.

#include <iostream>

int main()
{
   int sum{0};
   int count{};
   int x;
   while (std::cin >> x)
   {
      sum = sum + x;
      count = count + 1;
   }

   if (count != 0)
      std::cout << "average = " << sum / count << '\n';
}
```

Remember that `!=` is the C++ syntax for the ≠ operator. Thus `count != 0` is true when `count` is not zero, which means the program has read at least one number from its input.

Suppose you were to run the program with the following input:

```
2  5  3
```

What do you expect as the output?

Try it. **What is the actual output?**

Did you get what you expected? Some languages use different operators for integer division and floating-point division. C++ (like C) uses the same operator symbol and depends on the context to decide what kind of division to perform. If both operands are integers, the result is an integer.

What do you expect if the input is

```
2  5  4
```

Try it. **What is the actual output?**

Integer division truncates the result toward zero, so the C++ expression `5 / 3` equals `4 / 3` equals `1`.

The other arithmetic operators are `-` for subtraction, `*` for multiplication, and `%` for remainder. C++ does not have an operator for exponentiation.

Listing 3-3 asks for integers from the user and tells the user whether the number is even or odd. (Don’t worry about how input works in detail; Exploration 5 will cover that.) **Complete line 10**.

Listing 3-3. Testing for Even or Odd Integers

```
 1 /// Read integers and print a message that tells the user
 2 /// whether the number is even or odd.
 3
 4 #include <iostream>
 5
 6 int main()
 7 {
 8    int x;
 9    while (std::cin >> x)
10       if (                  )          // Fill in the condition.
11          std::cout << x << " is odd.\n";
12       else
13          std::cout << x << " is even.\n";
14 }
```

Test your program. **Did you get it right?**

I hope you used a line that looks something like this:

```
if (x % 2 != 0)
```

In other words, a number is odd if it has a non-zero remainder after dividing it by 2.

You know that `!=` compares for inequality. How do you think you should write an equality comparison? Try reversing the order of the odd and even messages, as shown in Listing 3-4. **Complete the condition on line 10**.

Listing 3-4. Testing for Even or Odd Integers

```
 1 /// Read integers and print a message that tells the user
 2 /// whether the number is even or odd.
 3
 4 #include <iostream>
 5
 6 int main()
 7 {
 8     int x;
 9     while (std::cin >> x)
10         if (                   )           // Fill in the condition.
11             std::cout << x << " is even.\n";
12         else
13             std::cout << x << " is odd.\n";
14 }
```

To test for equality, use two equal signs (==). In this case:

```
if (x % 2 == 0)
```

A common mistake that new C++ programmers make, especially those who are accustomed to SQL and similar languages, is to use a single equal sign for comparison. In this case, the compiler usually alerts you to the mistake. Go ahead and try it, to see what the compiler does. **What message does the compiler issue when you use a single equal sign in line 10?**

A single equal sign is the assignment operator. Thus, the C++ compiler thinks you are trying to assign the value 0 to the expression `x % 2`, which is nonsense, and the compiler rightly tells you so.

What if you want to test whether x is zero? **Modify Listing 3-1 to print a message when count is zero**. Once you get the program right, it should look something like Listing 3-5.

Listing 3-5. Print Average, Testing for a Zero Count

```
/// Read integers and print their average.
/// Print nothing if the input is empty.

#include <iostream>

int main()
{
    int sum{0};
    int count{};
    int x;
    while (std::cin >> x)
    {
        sum = sum + x;
        count = count + 1;
    }

17     if (count == 0)
18         std::cout << "No data.\n";
19     else
20         std::cout << "average = " << sum / count << '\n';
21 }
```

Now modify Listing 3-5 to use a single equal sign on line 17. **What message does your compiler issue?**

Most modern compilers recognize this common error and issue a warning. Strictly speaking, the code is correct: the condition assigns zero to count. Recall that a condition of zero means false, so the program always prints No data., regardless of how much data it actually reads.

If your compiler does not issue a warning, read the compiler's documentation. You might have to enable a switch to turn on extra warnings, such as "possible use of assignment instead of comparison" or "condition is always false."

As you can see, working with integers is easy and unsurprising. Text, however, is a little trickier, as you will see in the next Exploration.

EXPLORATION 4

Strings

In earlier Explorations, you used quoted character strings as part of each output operation. In this Exploration, you will begin to learn how to make your output a little fancier by doing more with strings. Start by reading Listing 4-1.

Listing 4-1. Different Styles of String Output

```
#include <iostream>

int main()
{
   std::cout << "Shape\tSides\n" << "-----\t-----\n";
   std::cout << "Square\t" << 4 << '\n' <<
                "Circle\t?\n";
}
```

Predict the output from the program in Listing 4-1. You may already know what \t means. If so, this prediction is easy to make. If you don't know, take a guess.

____

Now check your answer. Were you correct? **So what does \t mean?**

____

Inside a string, the backslash (\) is a special, even magical, character. It changes the meaning of the character that follows it. You have already seen how \n starts a new line. Now you know that \t is a horizontal tab: that is, it aligns the subsequent output at a tab position. In a typical console, tab stops are set every eight character positions.

How should you print a double-quote character in a string?

____

Write a program to test your hypothesis then run the program. **Were you correct?**

____

Compare your program with Listing 4-2.

Listing 4-2. Printing a Double-Quote Character

```
#include <iostream>

int main()
{
   std::cout << "\"\n";
}
```

In this case, the backslash turns a special character into a normal character. C++ recognizes a few other backslash character sequences, but these three are the most commonly used. (You'll learn a few more when you read about characters in Exploration 17.)

Now modify Listing 4-1 to add Triangle to the list of shapes.

What does the output look like? A tab character does not automatically align a column but merely positions the output at the next tab position. To align columns, you have to take control of the output. One easy way to do this is to use multiple tab characters, as shown in Listing 4-3.

Listing 4-3. Adding a Triangle and Keeping the Columns Aligned

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5    std::cout << "Shape\t\tSides\n" <<
 6                 "-----\t\t-----\n";
 7    std::cout << "Square\t\t" << 4 << '\n' <<
 8                 "Circle\t\t?\n"
 9                 "Triangle\t" << 3 << '\n';
10 }
```

I played a trick on you in Listing 4-3. Look closely at the end of line 8 and the start of line 9. Notice that the program lacks an output operator (<<) that ordinarily separates all output items. Any time you have two (or more) adjacent character strings, the compiler automatically combines them into a single string. This trick applies only to strings, not to characters. Thus, you can write lines 8 and 9 in many different ways, all meaning exactly the same thing.

```
std::cout << "\nCircle\t\t?\n" "Triangle\t" << 3 << '\n';
std::cout << "\nCircle\t\t?\nTriangle\t" << 3 << '\n';
std::cout << "\n" "Circle" "\t\t?\n" "Triangle" "\t" << 3 << '\n';
```

Choose the style you like best and stick with it. I like to make a clear break after each newline, so the human who reads my programs can clearly distinguish where each line ends and a new line begins.

You may be asking yourself why I bothered to print the numbers separately, instead of printing one big string. That's a good question. In a real program, printing a single string would be best, but in this book, I want to keep reminding you about the various ways you can write an output statement. Imagine, for example, what you would do if you didn't know beforehand the name of a shape and its number of sides. Perhaps that information is stored in variables, as shown in Listing 4-4.

Listing 4-4. Printing Information That Is Stored in Variables

```
 1 #include <iostream>
 2 #include <string>
 3
 4 int main()
 5 {
 6    std::string shape{"Triangle"};
 7    int sides{3};
 8
 9    std::cout << "Shape\t\tSides\n" <<
10                 "-----\t\t-----\n";
11    std::cout << "Square\t\t" << 4 << '\n' <<
12                 "Circle\t\t?\n";
13    std::cout << shape << '\t' << sides << '\n';
14 }
```

The type of a string is `std::string`. You must have `#include <string>` near the top of your program to inform the compiler that you are using the `std::string` type. Line 6 shows how to give an initial value to a string variable. Sometimes, you want the variable to start out empty. **How do you think you would define an empty string variable?**

____

____

Write a program to test your hypothesis.

If you have trouble verifying that the string is truly empty, try printing the string between two other, nonempty strings. Listing 4-5 shows an example.

Listing 4-5. Defining and Printing an Empty String

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6    std::string empty;
7    std::cout << "|" << empty << "|\n";
8 }
```

Compare your program with Listing 4-5. **Which do you prefer?** ________________

Why?

____

____

Line 6 does not provide an initial value for the variable, empty. You learned in Exploration 3 that omitting the initial value leaves the variable uninitialized, which would be an error, because no other value is assigned to empty. You are not allowed to print or otherwise access the value of an uninitialized variable. But `std::string` is different. The lack of an initializer in this case is the same as empty braces; namely, the variable is initialized to an empty string.

When you define a string variable with no initial value, C++ guarantees that the string is initially empty. **Modify Listing 4-4 so the shape and sides variables are uninitialized. Predict the output of the program**.

____

____

What happened? Explain.

____

____

____

Your program should look like Listing 4-6.

Listing 4-6. Demonstrating Uninitialized Variables

```
#include <iostream>
#include <string>

int main()
{
   std::string shape;
   int sides;

   std::cout << "Shape\t\tSides\n" <<
                "-----\t\t-----\n";
   std::cout << "Square\t\t" << 4 << '\n' <<
                "Circle\t\t?\n";
   std::cout << shape << '\t' << sides << '\n';
}
```

When I run Listing 4-6, I get different answers, depending on which compilers and platforms I use. Most compilers issue warnings, but will still compile the program, so you can run it. One of the answers I get is the following:

```
Shape              Sides
-----           -----
Square          4
Circle          ?
        4226851
```

With another compiler on another platform, the final number is 0. Yet another compiler's program prints -858993460 as the final number. Some systems might even crash instead of printing the value of shape or sides.

Isn't that curious? If you do not supply an initial value for a variable of type `std::string`, C++ makes sure the variable starts out with an initial value—namely, an empty string. On the other hand, if the variable has type int, you cannot tell what the initial value will actually be, and in fact, you cannot even tell whether the program will run. This is known as *undefined behavior*. The standard permits the C++ compiler and runtime environment to do anything, absolutely anything, when confronted with certain erroneous situations, such as accessing an uninitialized variable.

A design goal of C++ is that the compiler and library should not do any extra work if they can avoid it. Only the programmer knows what value makes sense as a variable's initial value, so assigning that initial value must be the programmer's responsibility. After all, when you are putting the finishing touches on your weather simulator (the one that will finally explain why it always rains when I go to the beach), you don't want the inner loop burdened by even one wasteful instruction. The flip side of that performance guarantee is an added burden on the programmer to avoid situations that give rise to undefined behavior. Some languages help the programmer avoid problem situations, but that help invariably comes with a performance cost.

So what's the deal with `std::string`? The short answer is that complicated types, such as strings, are different from the simple, built-in types. For types such as `std::string`, it is actually simpler for the C++ library to provide a well-defined initial value. Most of the interesting types in the standard library behave the same way.

If you have trouble remembering when it is safe to define a variable without an initial value, play it safe and use empty braces:

```
std::string empty{};

int zero{};
```

I recommend initializing every variable, even if you know the program will overwrite it soon, such as the input loops we used earlier. Omitting initialization in the name of "performance" rarely improves performance and always impairs readability. The next Exploration demonstrates the importance of initializing every variable.

OLD-FASHIONED INITIALIZATION

The brace style of initializing all variables was introduced in C++ 11, so code that predates C++ 11 (or new code that was written by programmers who learned C++ prior to C++ 11 and still haven't come to grips with the new style of initialization) uses different ways to initialize variables.

A common way to initialize integers, for example, uses an equal sign. It looks like an assignment statement, but it isn't. It defines and initializes a variable.

```
int x = 42;
```

You can also use parentheses:

```
int x(42);
```

The same is true for many standard-library types:

```
std::string str1 = "sample";
std::string str2("sample");
```

Other types require parentheses and don't work with an equal sign. Other types used curly braces before C++ 11. Equal signs, parentheses, and curly braces all had different rules, and it was hard for beginners to understand the subtle difference between the equal sign and parentheses for initialization.

So the standardization committee made an effort to define a single, uniform initialization style in C++ 11, which is what I presented in this Exploration.

EXPLORATION 5

Simple Input

So far, the Explorations have focused on output. Now it's time to turn your attention to input. Given that the output operator is <<, **what do you expect the input operator to be?** ________________

That didn't take a rocket scientist to deduce, did it? The input operator is >>, the opposite direction of the output operator. Think of the operators as arrows pointing in the direction that information flows: from the stream to variables for input, or from variables to the stream for output.

Listing 5-1 shows a simple program that performs input and output.

Listing 5-1. Demonstrating Input and Output

```
#include <iostream>

int main()
{
   std::cout << "Enter a number: ";
   int x;
   std::cin >> x;
   std::cout << "Enter another number: ";
   int y;
   std::cin >> y;

   int z{x + y};
   std::cout << "The sum of " << x << " and " << y << " is " << z << "\n";
}
```

How many numbers does Listing 5-1 read from the standard input? ________________

Suppose you enter 42 and 21 as the two input values. **What do you expect for the output**?

____

Now run the program and check your prediction. I hope you got 63. Suppose you type the following as input?

```
42*21
```

What do you predict will be the output?

____

Test your hypothesis. **What is the actual output?**

____

Do you see what happened? If not, try xyz as the input to the program. Try 42-21. Try 42.21.

The program exhibits two distinct behaviors that you have to understand. First, to read an int, the input stream must contain a valid integer. The integer can start with a sign (- or +) but must be all digits after that; no intervening whitespace is allowed. The input operation stops when it reaches the first character that cannot be part of a valid integer (such as *). If at least one digit is read from the input stream, the read succeeds, and the input text is converted to an integer. The read fails if the input stream does not start with a valid integer. If the read fails, the input variable is not modified.

The second behavior is what you discovered in the previous Exploration; uninitialized int variables result in undefined behavior. In other words, if a read fails, the variable contains junk, or worse. When you learn about floating point numbers, for example, you will learn that some bit patterns in an uninitialized floating-point variable can cause a program to terminate. On some specialized hardware, an uninitialized integer can do the same. The moral of the story is that using an uninitialized variable results in undefined behavior. That's bad. So don't do it.

Thus, when the input is xyz, both reads fail, and undefined behavior results. You probably see junk values for both numbers. When the input is 42-21, the first number is 42 and the second number is -21, so the result is correct. However, when the input is 42.21, the first number is 42, and the second number is junk, because an integer cannot start with a dot (.).

Once an input operation fails, all subsequent input attempts will also fail, unless you take remedial action. That's why the program doesn't wait for you to type a second number if the first one is invalid. C++ can tell you when an input operation fails, so your program can avoid using junk values. Also, you can reset a stream's error state, so you can resume reading after handling an error. I will cover these techniques in future Explorations. For now, make sure your input is valid and correct.

Some compilers warn you when your program leaves variables uninitialized, but it is best to be safe and initialize every variable, all the time. As you can see, even if the program immediately attempts to store a value in the variable, it might not succeed, which can give rise to unexpected behavior.

Did you think that integers could be so complicated? Surely strings are simpler, because there is no need to interpret them or convert their values. Let's see if they truly are simpler than integers. Listing 5-2 is similar to Listing 5-1, but it reads text into std::string variables.

Listing 5-2. Reading Strings

```
#include <iostream>
#include <string>

int main()
{
   std::cout << "What is your name? ";
   std::string name{};
   std::cin >> name;
   std::cout << "Hello, " << name << ", how are you? ";
   std::string response{};
   std::cin >> response;
   std::cout << "Good-bye, " << name << ". I'm glad you feel " << response << "\n";
}
```

Listing 5-2 is clearly not a model of artificial intelligence, but it demonstrates one thing well. Suppose the input is as follows:

```
Ray Lischner
Fine
```

What do you expect as the output?

____

____

____

Run the program and test your hypothesis. **Were you correct?** ____________

Explain.

____

____

Experiment with different input and try to discern the rules that C++ uses to delimit a string in the input stream. Ready? Go ahead. I'll wait until you're done.

Back so soon? **How does C++ delimit strings in an input stream?**

____

____

____

Any whitespace character (the exact list of whitespace characters depends on your implementation, but typically includes blanks, tabs, newlines, and the like) ends a string, at least as far as the input operation is concerned. Specifically, C++ skips leading whitespace characters. Then it accumulates nonspace characters to form the string. The string ends at the next whitespace character.

So what happens when you mix integers and strings? **Write a program that asks for a person's name (first name only) and age (in years) and then echoes the input to the standard output.** Which do you want to ask for first? Print the information after reading it.

Table 5-1 shows some sample inputs for your program. Next to each one, **write your prediction for the program's output.** Then run the program, and **write the actual output.**

Table 5-1. *Sample Inputs for Name and Age*

Input	Predicted Output	Actual Output
`Ray44`		
`44Ray`		
`Ray 44`		
`44 Ray`		
`Ray 44`		
`44 Ray`		
`44-Ray`		
`Ray-44`		

Think of the standard input as a stream of characters. Regardless of how the user types those characters, the program sees them arrive one by one. (Okay, they arrive in big chunks, by the buffer-load, but that's a minor implementation detail. As far as you are concerned, your program reads one character at a time, and it doesn't matter that the character comes from the buffer, not the actual input device.) Thus, the program always maintains the notion of a current position in the stream. The next read operation always starts at that position.

Before starting any input operation, if the character at the input position is a whitespace character, the program skips (that is, reads and discards) that character. It keeps reading and discarding characters until it reaches a nonspace character. Then the actual read begins.

If the program attempts to read an integer, it grabs the character at the input position and checks whether it is valid for an integer. If not, the read fails, and the input position does not move. Otherwise, the input operation keeps the character and all subsequent characters that are valid elements of an integer. The input operation interprets the text as an integer and stores the value in the variable. Thus, after reading an integer, you know that the input position points to a character that is *not* a valid integer character.

When reading a string, all the characters are grabbed from the stream until a whitespace character is reached. Thus, the string variable does not contain any whitespace characters. The next read operation will skip over the whitespace, as described earlier.

The input stream ends at the end of the file (if reading from a file), when the user closes the console or terminal, or when the user types a special keystroke sequence to tell the operating system to end the input (such as Control+D on UNIX or Control+Z on DOS or Windows). Once the end of the input stream is reached, all subsequent attempts to read from the stream will fail. This is what caused the loop to end in Exploration 2.

Listing 5-3 shows my version of the name-first program. Naturally, your program will differ in the details, but the basic outline should agree with yours.

Listing 5-3. Getting the User's Name and Age

```
#include <iostream>
#include <string>

int main()
{
   std::cout << "What is your name? ";
   std::string name{};
   std::cin >> name;

   std::cout << "Hello, " << name << ", how old are you? ";
   int age{};
   std::cin >> age;

   std::cout << "Good-bye, " << name << ". You are " << age << " year";
   if (age != 1)
      std::cout << 's';
   std::cout << " old.\n";
}
```

Now modify the program to reverse the order of the name and age and try all the input values again. Explain what you observe.

____

____

____

When an input operation fails due to malformed input, the stream enters an error state; e.g., the input stream contains the string "Ray" when the program tries to read an integer. All subsequent attempts to read from the stream result in an error, without actually trying to read. Even if the stream subsequently tries to read a string, which would otherwise succeed, the error state is sticky, and the string read fails too.

In other words, when the program cannot read the user's age, it won't be able to read the name either. That's why the program gets both name and age correct, or it gets both wrong.

Listing 5-4 shows my version of the age-first program.

Listing 5-4. Getting the User's Age and Then Name

```
#include <iostream>
#include <string>

int main()
{
   std::cout << "How old are you? ";
   int age{};
   std::cin >> age;

   std::cout << "What is your name? ";
   std::string name{};
   std::cin >> name;

   std::cout << "Good-bye, " << name << ". You are " << age << " year";
   if (age != 1)
      std::cout << 's';
   std::cout << " old.\n";
}
```

Table 5-2 shows a truncated version of the output (just the name and age) in each situation.

Table 5-2. *Interpreting Input the C++ Way*

Input	Name First	Age First
`Ray44`	`"Ray44", 0`	`0, ""`
`44Ray`	`"44Ray", 0`	`44, "Ray"`
`Ray 44`	`"Ray", 44`	`0, ""`
`44 Ray`	`"44", 0`	`44, "Ray"`
`Ray 44`	`"Ray", 44`	`0, ""`
`44 Ray`	`"44", 0`	`44, "Ray"`
`44#Ray`	`"44#Ray", 0`	`44, "#Ray"`
`Ray#44`	`"Ray#44", 0`	`0, ""`

Handling errors in an input stream requires some more advanced C++, but handling errors in your code is something you can take care of right now. The next Exploration helps you untangle compiler error messages.

EXPLORATION 6

Error Messages

By now you've seen plenty of error messages from your C++ compiler. No doubt, some are helpful and others are cryptic—a few are both. This Exploration presents a number of common errors and gives you a chance to see what kinds of messages your compiler issues for these mistakes. The more familiar you are with these messages, the easier it will be for you to interpret them in the future.

Read through Listing 6-1 and keep an eye out for mistakes.

Listing 6-1. Deliberate Errors

```
#include <iosteam>
// Look for errors
int main()
[
  std::cout < "This program prints a table of squares.\n";
         "Enter the starting value for the table: ";
  int start{0};
  std::cin >> start;
  std::cout << "Enter the ending value for the table: ";
  int end(start);
  std::cin << endl
  std::cout << "#   #^2\n";
  int x{start};
  end = end + 1; // exit loop when x reaches end
  while (x != end)
  {
    std:cout << x << "   " << x*x << "\n";
    x = x + 1;
  }
}
```

What errors do you expect the compiler to detect?

Download the source code and compile Listing 6-1.

What messages does your compiler actually issue?

____

____

____

____

____

____

____

Create three groups: messages that you correctly predicted, messages that you expected but the compiler did not issue, and messages that the compiler issued but you did not expect. **How many messages are in each group?**

____

If you use command-line tools, expect to see a slew of errors running across your screen. If you use an IDE, it will help corral the error messages and associate each message with the relevant point in the source code that the compiler thinks is the cause of the error. The compiler isn't always right, but its hint is often a good starting point.

Compilers usually group problems into one of two categories: errors and warnings. An error prevents the compiler from producing an output file (an object file or program). A warning tells you that something is wrong but doesn't stop the compiler from producing its output. Modern compilers are pretty good at detecting problematic, but valid, code and issuing warnings, so get in the habit of heeding the warnings. In fact, I recommend dialing up the sensitivity to warnings. Check your compiler's documentation and look for options that have the compiler detect as many warnings as possible. For g++ and clang++, the switch is `-Wall`. Visual Studio uses `/Wall`. On the other hand, sometimes the compiler gets it wrong, and certain warnings are not helpful. You can usually disable individual warnings, such as `-Wno-unused-local-typedefs` (my favorite for g++) or `/wd4514` (for Visual Studio).

The program actually contains seven errors, but don't fret if you missed them. Let's take them one at a time.

Misspelling

Line 1 misspells `<iostream>` as `<iosteam>`. Your compiler should give you a simple message, informing you that it could not find `<iosteam>`. The compiler probably cannot tell that you meant to type `<iostream>`, so it does not give you a suggestion. You have to know the proper spelling of the header name.

Most compilers give up completely at this point. If that happens to you, fix this one error then run the compiler again, to see some more messages.

If your compiler tries to continue, it does so without the declarations from the misspelled header. In this case, `<iostream>` declares `std::cin` and `std::cout`, so the compiler also issues messages about those names being unknown, as well as other error messages about the input and output operators.

Bogus Character

The most interesting error is the use of a square-bracket character ([) instead of a brace character ({) in line 4. Some compilers may be able to guess what you meant, which can limit the resulting error messages. Others cannot and give a message that may be rather cryptic. For example, g++ issues many errors, none of which directly points you to the error. Instead, it issues the following messages:

```
list0601.cpp:5:58: error: array bound is not an integer constant before ';' token


   std::cout < "This program prints a table of squares.\n";
                                                          ^
list0601.cpp:5:58: error: expected ']' before ';' token
list0601.cpp:6:10: error: expected unqualified-id before string constant
          "Enter the starting value for the table: ";
          ^
list0601.cpp:8:3: error: 'cin' in namespace 'std' does not name a type
   std::cin >> start;
   ^
list0601.cpp:9:3: error: 'cout' in namespace 'std' does not name a type
   std::cout << "Enter the ending value for the table: ";
   ^
list0601.cpp:11:3: error: 'cin' in namespace 'std' does not name a type
   std::cin << endl
   ^
list0601.cpp:14:3: error: 'end' does not name a type
   end = end + 1; // exit loop when x reaches end
   ^
list0601.cpp:15:3: error: expected unqualified-id before 'while'
   while (x != end)
   ^
list0601.cpp:20:1: error: expected declaration before '}' token
 }
 ^
```

When you cannot understand the error messages, look at the first message and the line number it identifies. Search for errors at or near the line number. Ignore the rest of the messages.

On line 5, you may see another error or two. After you fix them, however, a slew of messages still remains. That means you still haven't found the real culprit (which is on line 4). A different compiler (clang++) is more helpful. It points you to line 4.

```
list0601.cpp:5:59: error: expected ']'
  std::cout < "This program prints a table of squares.\n";
                                                          ^
list0601.cpp:4:1: note: to match this '['
[
^
```

Once you track down the square bracket and change it to a curly brace, you may get entirely different messages. This is because the substitution of [for { so thoroughly confuses the compiler, it cannot make any sense of the rest of the program. Correcting that problem straightens out the program for the compiler, but now it may find a whole new set of errors.

Unknown Operator

The input and output operators (>> and <<) are no different from any other C++ operator, such as addition (+), multiplication (*), or comparison (such as >). Every operator has a limited set of allowed operands. For example, you cannot "add" two I/O streams (e.g., `std::cin + std::cout`), nor can you use an output operator to "write" a number to a string (e.g., `"text" << 3`).

On line 5, one error is the use of < instead of <<. The compiler cannot determine that you intended to use << and instead issues a message that indicates what is wrong with <. The exact message depends on the compiler, but most likely the message is not something that helps you solve this particular problem. One compiler complains as follows:

```
list0601.cxx: In function 'int main()':
list0601.cxx:5: error: no match for 'operator<' in 'std::cout < "This program➥
 prints a table of squares.\012"'
list0601.cxx:5: note: candidates are: operator<(const char*, const char*) <built-in>
list0601.cxx:5: note:                 operator<(void*, void*) <built-in>
```

This message notifies you that you are using the wrong operator or the wrong operands. You must determine which one it is.

Once you fix the operator, notice that the compiler does not issue any message for the other mistake, namely, the extraneous semicolon. Strictly speaking, it is not a C++ error. It is a logical error, but the result is a valid C++ program. Some compilers will issue a warning, advising you that line 6 does nothing, which is a hint that you made a mistake. Other compilers will silently accept the program.

The only sure way to detect this kind of mistake is to learn to proofread your code.

Unknown Name

An easy error for a compiler to detect is when you use a name that the compiler does not recognize at all. In this case, accidentally typing the letter `l` instead of a semicolon produces the name `endl` instead of `end;`. The compiler issues a clear message about this unknown name.

Fix the semicolon, and now the compiler complains about another operator. This time, you should be able to zoom in on the problem and notice that the operator is facing the wrong way (<< instead of >>). The compiler may not offer much help, however. One compiler spews out errors of the following form:

```
list0601.cxx
list0601.cxx(11) : error C2784: 'std::basic_ostream<char,_Traits> &std::operator➥
<<(std::basic_ostream<char,_Traits> &,unsigned char)' : could not deduce➥
template argument for 'std::basic_ostream<char,_Elem> &' from 'std::istream'
        C:\Program Files\Microsoft Visual C++ Toolkit 2003\include\ostream(887)➥
: see declaration of 'std::operator'<<''
list0601.cxx(11) : error C2784: 'std::basic_ostream<char,_Traits> &std::operator➥
<<(std::basic_ostream<char,_Traits> &,unsigned char)' : could not deduce➥
template argument for 'std::basic_ostream<char,_Elem> &' from 'std::istream'
        C:\Program Files\Microsoft Visual C++ Toolkit 2003\include\ostream(887)➥
: see declaration of 'std::operator'<<''
```

The line number tells you where to look, but it is up to you to find the problem.

Symbol Errors

But now you run into a strange problem. The compiler complains that it does not know what a name means (`cout` on line 17), but you know what it means. After all, the rest of the program uses `std::cout` without any difficulty. What's wrong with line 17 that it causes the compiler to forget?

Small errors can have profound consequences in C++. As it turns out, a single colon means something completely different from a double colon. The compiler sees `std:cout` as a statement labeled `std`, followed by the bare name `cout`. At least the error message points you to the right place. Then it's up to you to notice the missing colon.

Fun with Errors

After you have fixed all the syntax and semantic errors, compile and run the program, to make sure you truly found them all. Then introduce some new errors, just to see what happens. Some suggestions follow:

Try dropping a semicolon from the end of a statement. What happens?

____

____

Try dropping a double quote from the start or end of a string. What happens?

____

____

Try misspelling int as iny. What happens?

____

____

Now I want you to explore on your own. Introduce one error at a time and see what happens. Try making several errors at once. Sometimes, errors have a way of obscuring one another. Go wild! Have fun! How often does your teacher encourage you to make mistakes?

Now it's time to return to correct C++ code. The next Exploration introduces the `for` loop.

EXPLORATION 7

For Loops

Explorations 2 and 3 show some simple `while` loops. This Exploration introduces the `while` loop's big brother, the `for` loop.

Bounded Loops

You've already seen `while` loops that read from the standard input until no more input is available. That is a classic case of an *unbounded* loop. Unless you know beforehand exactly what the input stream will contain, you cannot determine the loop's bounds or limits. Sometimes you know in advance how many times the loop must run; that is, you know the bounds of the loop, making it a *bounded* loop. The `for` loop is how C++ implements a bounded loop.

Let's start with a simple example. Listing 7-1 shows a program that prints the first ten non-negative integers.

Listing 7-1. Using a `for` Loop to Print Ten Non-Negative Numbers

```
#include <iostream>

int main()
{
  for (int i{0}; i != 10; i = i + 1)
    std::cout << i << '\n';
}
```

The `for` loop crams a lot of information in a small space, so take it one step at a time. Inside the parentheses are three parts of the loop, separated by semicolons. **What do you think these three pieces mean?**

____

____

____

____

The three parts are: initialization, condition, and postiteration. Take a closer look at each part.

Initialization

The first part looks similar to a variable definition. It defines an `int` variable named `i`, with an initial value of 0. Some C-inspired languages permit only an initialization expression, not a variable definition. In C++, you have a choice: expression or definition. The advantage of defining the loop control variable as part of the initialization is that you cannot accidentally refer to that variable outside the loop. The disadvantage of defining the loop control variable in the initialization part is that you cannot deliberately refer to that variable outside the loop. Listing 7-2 demonstrates the advantage of limiting the loop control variable.

Listing 7-2. You Cannot Use the Loop Control Variable Outside the Loop

```
#include <iostream>

int main()
{
  for (int i{0}; i != 10; i = i + 1)
    std::cout << i << '\n';
  std::cout << "i=" << i << '\n';        // error: i is undefined outside the loop
}
```

Another consequence of limiting the loop control variable is that you may define and use the same variable name in multiple loops, as shown in Listing 7-3.

Listing 7-3. Using and Reusing a Loop Control Variable Name

```
#include <iostream>

int main()
{
  std::cout << '+';
  for (int i{0}; i != 20; i = i + 1)
    std::cout << '-';
  std::cout << "+\n|";

  for (int i{0}; i != 3; i = i + 1)
    std::cout << ' ';
  std::cout << "Hello, reader!";

  for (int i{0}; i != 3; i = i + 1)
    std::cout << ' ';
  std::cout << "|\n+";

  for (int i{0}; i != 20; i = i + 1)
    std::cout << '-';
  std::cout << "+\n";
}
```

What does Listing 7-3 produce as output?

____

____

____

If you don't have to perform any initialization, you can leave the initialization part empty, but you still need the semicolon that separates the empty initialization from the condition.

Condition

The middle part follows the same rules as a `while` loop condition. As you might expect, it controls the loop execution. The loop body executes while the condition is true. If the condition is false, the loop terminates. If the condition is false the first time the loop runs, the loop body never executes (but the initialization part always does).

Sometimes you will see a `for` loop with a missing condition. That means the condition is always true, so the loop runs without stopping. A better way to write a condition that is always true is to be explicit and use `true` as the condition. That way, anyone who has to read and maintain your code in the future will understand that you deliberately designed the loop to run forever. Think of it as the equivalent of a comment: "This condition deliberately left blank."

Postiteration

The last part looks like a statement, even though it lacks the trailing semicolon. In fact, it is not a full statement, but only an expression. The expression is evaluated after the loop body (hence the name *post*iteration) and before the condition is tested again. You can put anything you want here, or leave it blank. Typically, this part of the `for` loop controls the iteration, advancing the loop control variable as needed.

How a for `Loop` Works

The flow of control is as follows:

1. The initialization part runs exactly once.
2. The condition is tested. If it is false, the loop terminates, and the program continues with the statement that follows the loop body.
3. If the condition is true, the loop body executes.
4. The postiteration part executes.
5. Control jumps to 2.

Your Turn

Now it's your turn to write a `for` loop. Listing 7-4 shows a skeleton of a C++ program. **Fill in the missing parts to compute the sum of integers from 10 to 20, inclusive.**

Listing 7-4. Compute Sum of Integers from 10 to 20

```
#include <iostream>

int main()
{
  int sum{0};

  // Write the loop here.

  std::cout << "Sum of 10 to 20 = " << sum << '\n';
}
```

Before you test your program, you must first determine how you will know whether the program is correct. In other words, **what is the sum of the integers from 10 to 20, inclusive?** ________________

Okay, now compile and run your program. **What answer does your program produce?** ________________ **Is your program correct?** ________________

Compare your program with that shown in Listing 7-5.

Listing 7-5. Compute Sum of Integers from 10 to 20 (Completed)

```
#include <iostream>

int main()
{
  int sum{0};
  for (int i{10}; i != 21; i = i + 1)
    sum = sum + i;
  std::cout << "Sum of 10 to 20 = " << sum << '\n';
}
```

A use of `for` loops is to format and print tables of information. To accomplish this, you need finer control over output formatting than what you have learned so far. That will be the subject of the next Exploration.

EXPLORATION 8

Formatted Output

In Exploration 4, you used tab characters to line up output neatly. Tabs are useful but crude. This Exploration introduces some of the features that C++ offers to format output nicely, such as setting the alignment, padding, and width of output fields.

The Problem

This Exploration begins a little differently. Instead of reading a program and answering questions about it, you must write your own program to solve a problem. The task is to print a table of squares and cubes (the mathematical variety, not the geometrical shapes) for integers from 1 to 20. The output of the program should look something like the following:

```
 N     N^2     N^3
 1       1       1
 2       4       8
 3       9      27
 4      16      64
 5      25     125
 6      36     216
 7      49     343
 8      64     512
 9      81     729
10     100    1000
11     121    1331
12     144    1728
13     169    2197
14     196    2744
15     225    3375
16     256    4096
17     289    4913
18     324    5832
19     361    6859
20     400    8000
```

To help you get started, Listing 8-1 gives you a skeleton program. You need only fill in the loop body.

Listing 8-1. Print a Table of Squares and Cubes

```
#include <iomanip>
#include <iostream>

int main()
{
  std::cout << " N   N^2    N^3\n";
  for (int i{1}; i != 21; ++i)
  {
    // write the loop body here
  }
}
```

This is a trick problem, so don't worry if you had difficulties. The point of this exercise is to demonstrate how difficult formatted output actually is. If you've learned that much, you successfully completed this exercise, even if you didn't finish the program. Perhaps you tried using tab characters at first, but that aligns the numbers on the left.

```
N       N^2     N^3
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

Left-alignment is not the way we usually write numbers. Tradition dictates that numbers should align to the right (or on decimal points, when applicable—more on that in the related section, "Alignment," later in this Exploration). Right-aligned numbers are easier to read.

C++ offers some simple but powerful techniques to format output. To format the table of powers, you have to define a field for each column. A field has a width, a fill character, and an alignment. The following sections explain these concepts in depth.

Field Width

Before exploring how you would specify alignment, first you must know how to set the width of an output field. I gave you a hint in Listing 8-1. **What is the hint?**

____

The first line of the program is `#include <iomanip>`, which you have not seen before. This header declares some useful tools, including `std::setw()`, which sets the minimum width of an output field. For example, to print a number so that it occupies at least three character positions, call `std::setw(3)`. If the number requires more space than that—say the value is 314159—the actual output will take up as much space as needed. In this case, the spacing turned out to be six character positions.

To use `setw`, call the function as part of an output statement. The statement looks like you are trying to print `setw`, but in fact, nothing is printed, and all you are doing is manipulating the state of the output stream. That's why `setw` is called an *I/O manipulator*. The `<iomanip>` header declares several manipulators, which you will learn about in due course.

Listing 8-2 shows the table of powers program, using `setw` to set the width of each field in the table.

Listing 8-2. Printing a Table of Powers the Right Way

```
#include <iomanip>
#include <iostream>

int main()
{
  std::cout << " N   N^2    N^3\n";
  for (int i{1}; i != 21; ++i)
    std::cout << std::setw(2) << i
              << std::setw(6) << i*i
              << std::setw(7) << i*i*i
              << '\n';
}
```

The first column of the table requires two positions, to accommodate numbers up to 20. The second column needs some space between columns and room for numbers up to 400; `setw(6)` uses three spaces between the `N` and the `N^2` columns and three character positions for the number. The final column also uses three spaces between columns and four character positions, to allow numbers up to 8000.

The default field width is zero, which means everything you print takes up the exact amount of space it needs, no more, no less.

After printing one item, the field width automatically resets to zero. For example, if you wanted to use a uniform column width of six for the entire table, you could not call `setw(6)` once and leave it at that. Instead, you must call `setw(6)` before each output operation, as follows:

```
    std::cout << std::setw(6) << i
              << std::setw(6) << i*i
              << std::setw(6) << i*i*i
              << '\n';
```

Fill Character

By default, values are padded, or filled, with space characters (`' '`). You can set the fill character to be any that you choose, such as zero (`'0'`) or an asterisk (`'*'`). Listing 8-3 shows a fanciful use of both fill characters in a program that prints a check.

Listing 8-3. Using Alternative Fill Characters

```
#include <iomanip>
#include <iostream>

int main()
{
  using namespace std;

  int day{14};
  int month{3};
  int year{2006};
  int dollars{42};
  int cents{7};
  // Print date in USA order. Later in the book, you will learn how to
  // handle internationalization.
  cout << "Date: " << setfill('0') << setw(2) << month
                          << '/' << setw(2) << day
                          << '/' << setw(2) << year << '\n';
  cout << "Pay to the order of: CASH\n";
  cout << "The amount of $" << setfill('*') << setw(8) << dollars << '.'
                          << setfill('0') << setw(2) << cents << '\n';
}
```

Notice that unlike `setw`, `setfill` is sticky. That is, the output stream remembers the fill character and uses that character for all output fields until you set a different fill character.

`std` Prefix

Another new feature in Listing 8-3 is the declaration `using namespace std;`. All those `std::` prefixes can sometimes make the code hard to read. The important parts of the names become lost in the clutter. By starting your program with `using namespace std;`, you are instructing the compiler to treat names that it doesn't recognize as though they began with `std::`.

As the keyword indicates, `std` is called a *namespace*. Almost every name in the standard library is part of the `std` namespace. You are not allowed to add anything to the `std` namespace, nor are any third-party library vendors. Thus, if you see `std::`, you know that what follows is part of the standard library (so you can look it up in any reliable reference). More important, you know that most names you invent in your own program will not conflict with any name in the standard library, and vice versa. Namespaces keep your names separate from the standard library names. Later in this book, you will learn to create your own namespaces, which help organize libraries and manage large applications.

On the other hand, `using namespace std;` is a dangerous declaration, and one I use sparingly. Without the `std::` qualifier before every standard library name, you have opened the door to confusion. Imagine, for example, if your program defines a variable named `cout` or `setw`. The compiler has strict rules for interpreting names and would not be confused at all, but the human reader certainly would be. It is always best to avoid names that collide with those in the standard library, with or without `using namespace std;`.

Alignment

C++ lets you align output fields to the right or the left. If you want to center a number, you are on your own. To force the alignment to be left or right, use the `left` and `right` manipulators, which you get for free when you include `<iostream>`. (The only time you need `<iomanip>` is when you want to use manipulators that require additional information, such as `setw` and `setfill`.)

The default alignment is to the right, which might strike you as odd. After all, the first attempt at using tab characters to align the table columns produced left-aligned values. As far as C++ is concerned, however, it knows nothing about your table. Alignment is within a field. The `setw` manipulator specifies the width, and the alignment determines whether the fill characters are added on the right (left-alignment) or on the left (right-alignment). The output stream has no memory of other values it may have printed earlier (such as on the previous line). So, for example, if you want to align a column of numbers on their decimal points, you must do that by hand (or ensure that every value in the column has the same number of digits after the decimal point).

Exploring Formatting

Now that you know the rudiments of formatting output fields, it is time to explore a little and help you develop a thorough understanding of how field width, fill character, and alignment interact. **Read the program in Listing 8-4 and predict its output.**

Listing 8-4. Exploring Field Width, Fill Character, and Alignment

```
#include <iomanip>
#include <iostream>

int main()
{
  using namespace std;
  cout << '|' << setfill('*') << setw(6) <<  1234 << '|' << '\n';
  cout << '|' << left <<         setw(6) <<  1234 << '|' << '\n';
  cout << '|' <<                 setw(6) << -1234 << '|' << '\n';
  cout << '|' << right <<        setw(6) << -1234 << '|' << '\n';
}
```

What do you expect as the output from Listing 8-4?

____

____

____

____

Now write a program that will produce the following output. Don't cheat and simply print a long string. Instead, print only integers and newlines, throwing in the field width, fill character, and alignment manipulators you require to achieve the desired output.

```
000042
420000
42
-42-
```

Lots of different programs can achieve the same goal. My program, shown in Listing 8-5, is only one possibility of many.

Listing 8-5. Program to Produce Formatted Output

```
#include <iomanip>
#include <iostream>

int main()
{
  using namespace std;

  cout << setfill('0') << setw(6) << 42 << '\n';
  cout << left         << setw(6) << 42 << '\n';
  cout << 42 << '\n';
  cout << setfill('-') << setw(4) << -42 << '\n';
}
```

The manipulators that take arguments, such as `setw` and `setfill`, are declared in `<iomanip>`. The manipulators without arguments, such as `left` and `right`, are declared in `<iostream>`. If you can't remember, include both headers. If you include a header that you don't really need, you might see a slightly slower compilation time, but no other ill effects will befall you.

I'M LYING TO YOU

The `left` and `boolalpha` manipulators are *not* declared in `<iostream>`. I lied to you. They are actually declared in `<ios>`. But `<iostream>` contains `#include <ios>`, so you get everything in `<ios>` automatically when you include `<iostream>`.

I've been lying to you for some time. The input operator (`>>`) is actually declared in `<istream>`, and the output operator (`<<`) is declared in `<ostream>`. As with `<ios>`, the `<iostream>` header always includes `<istream>` and `<ostream>`. Thus, you can include `<iostream>` and get all the headers you need for typical input and output. Other headers, such as `<iomanip>`, are less commonly used, so they are not part of `<iostream>`.

So I wasn't really lying to you, just waiting until you could handle the truth.

Alternative Syntax

I like to use manipulators, because they are concise, clear, and easy to employ. You can also apply functions to the output stream object, using the dot operator (`.`). For example, to set the fill character, you can call `std::cout.fill('*')`. The `fill` function is called a *member function,* because it is a member of the output stream's type. You cannot apply it to any other kind of object. Only some types have member functions, and each type defines the member functions that it allows. A large part of any C++ library reference is taken up with the various types and their member functions. (The member functions of an output stream are declared in `<ostream>`, along with the output operators. An input stream's member functions are declared in `<istream>`. Both of these headers are included automatically when you include `<iostream>`.)

When setting sticky properties, such as fill character or alignment, you might prefer using member functions instead of manipulators. You can also use member functions to query the current fill character, alignment and other flags, and field width—something you can't do with manipulators.

The member function syntax uses the stream object, a dot (`.`), and the function call, e.g., `cout.fill('0')`. Setting the alignment is a little more complicated. Listing 8-6 shows the same program as Listing 8-5 but uses member functions instead of manipulators.

Listing 8-6. A Copy of Listing 8-5, but Using Member Functions

```
#include <iostream>

int main()
{
  using namespace std;

  cout.fill('0');
  cout.width(6);
  cout << 42 << '\n';
  cout.setf(ios_base::left, ios_base::adjustfield);
  cout.width(6);
  cout << 42 << '\n';
  cout << 42 << '\n';
  cout.fill('-');
  cout.width(4);
  cout << -42 << '\n';
}
```

To query the current fill character, call `cout.fill()`. That's the same function name you use to set the fill character, but when you call the function with no arguments, it returns the current fill character. Similarly, `cout.width()` returns the current field width. Obtaining the flags is slightly different. You call `setf` to set flags, such as the alignment, but you call `flags()` to return the current flags. The details are not important at this time, but if you're curious, consult any relevant library reference.

On Your Own

Now it is time for you to write a program from scratch. Feel free to look at other programs, to make sure you have all the necessary parts. Write this program to produce a multiplication table for the integers from 1 to 10, inclusive, as follows:

```
   *|   1   2   3   4   5   6   7   8   9  10
----+----------------------------------------
   1|   1   2   3   4   5   6   7   8   9  10
   2|   2   4   6   8  10  12  14  16  18  20
   3|   3   6   9  12  15  18  21  24  27  30
   4|   4   8  12  16  20  24  28  32  36  40
   5|   5  10  15  20  25  30  35  40  45  50
   6|   6  12  18  24  30  36  42  48  54  60
   7|   7  14  21  28  35  42  49  56  63  70
   8|   8  16  24  32  40  48  56  64  72  80
   9|   9  18  27  36  45  54  63  72  81  90
  10|  10  20  30  40  50  60  70  80  90 100
```

After you finish your program and have made sure it produces the correct output, compare your program with mine, which is shown in Listing 8-7.

Listing 8-7. Printing a Multiplication Table

```
#include <iomanip>
#include <iostream>

int main()
{
  using namespace std;

  int const low{1};        ///< Minimum value for the table
  int const high{10};      ///< Maximum value for the table
  int const colwidth{4};   ///< Fixed width for all columns

  // All numbers must be right-aligned.
  cout << right;
  // First print the header.
  cout << setw(colwidth) << '*'
       << '|';
  for (int i{low}; i <= high; i = i + 1)
    cout << setw(colwidth) << i;
  cout << '\n';

  // Print the table rule by using the fill character.
  cout << setfill('-')
       << setw(colwidth) << ""                    // one column's worth of "-"
       << '+'                                     // the vert. & horz. intersection
       << setw((high-low+1) * colwidth) << ""     // the rest of the line
       << '\n';

  // Reset the fill character.
  cout << setfill(' ');

  // For each row...
  for (int row{low}; row <= high; row = row + 1)
  {
    cout << setw(colwidth) << row << '|';
    // Print all the columns.
    for (int col{low}; col <= high; col = col + 1)
      cout << setw(colwidth) << row * col;
    cout << '\n';
  }
}
```

My guess is that you wrote your program a little differently from how I wrote mine, or perhaps you wrote it very differently. That's okay. Most likely, you used a hard-coded string for the table rule (the line that separates the header from the table), or perhaps you used a `for` loop. I used I/O formatting, just to show you what is possible. Printing an empty string with a nonzero field width is a quick and easy way to print a repetition of a single character.

Another new feature I threw in for good measure is the `const` keyword. Use this keyword in a definition to define the object as a constant instead of a variable. The compiler makes sure you do not accidentally assign anything to the object. As you know, named constants make programs easier to read and understand than littering the source code with numbers.

Loops sure are fun! What data structure do you think of first when you think of loops? I hope you picked arrays, because that is the subject of the next Exploration.

EXPLORATION 9

Arrays and Vectors

Now that you understand the basics, it is time to start moving on to more exciting challenges. Let's write a real program, something nontrivial but still simple enough to master this early in the book. Your job is to write a program that reads integers from the standard input, sorts them into ascending order, and then prints the sorted numbers, one per line.

At this point, the book has not quite covered enough material for you to solve this problem, but it is instructive to think about the problem and the tools you may need to solve it. Your first task in this Exploration is to **write pseudo-code for the program.** Write C++ code where you can and make up whatever you need to tackle the problem.

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

Vectors for Arrays

You need an array to store the numbers. Given only that much new information, you can write a program to read, sort, and print numbers, but only by hand-coding the sort code. Those of you who suffered through a college algorithms course may remember how to write a bubble sort or quick sort, but why should you need to muck about with such low-level code? Surely, you say, there's a better way. There is: the C++ standard library has a fast sort function that can sort just about anything. Jump straight into the solution in Listing 9-1.

Listing 9-1. Sorting Integers

```
 1 #include <algorithm>
 2 #include <iostream>
 3 #include <vector>
 4
 5 int main()
 6 {
 7   std::vector<int> data{};     // initialized to be empty
 8   int x{};
 9
10   // Read integers one at a time.
11   while (std::cin >> x)
12     // Store each integer in the vector.
13     data.push_back(x);
14
15   // Sort the vector.
16   std::sort(data.begin(), data.end());
17
18   // Print the vector, one number per line.
19   for (std::vector<int>::size_type i{0}, end{data.size()}; i != end; ++i)
20     std::cout << data.at(i) << '\n';
21 }
```

Notice anything unusual about the program? Where is the array? C++ has a type called `vector`, which is a resizable array type. The next section explains it all to you.

Vectors

Line 7 defines the variable `data`, of type `std::vector<int>`. C++ has several container types, that is, data structures that can contain a bunch of objects. One of those containers is `vector`, which is an array that can change size. All C++ containers require an element type, that is, the type of object that you intend to store in the container. In this case, the element type is `int`. Specify the element type in angle brackets: `<int>`. That tells the compiler that you want data to be a `vector` and that the `vector` will store integers.

What's missing from the definition?

The vector has no size. Instead, the vector can grow or shrink while the program runs. (If you know that you need an array of a specific, fixed size, you can use the type `array`. You will use `vector` much more often than `array`.) Thus, `data` is initially empty. Like `std::string`, `vector` is a library type, and it has a well-defined initial value, namely, empty, so you can omit the `{}` initializer if you wish.

You can insert and erase items at any position in the vector, although the performance is best when you add items only at the end or erase them only from the end. That's how the program stores values in `data`: by calling `push_back`, which adds an element to the end of a `vector` (line 13). The "back" of a vector is the end, with the highest index. The "front" is the beginning, so `back()` returns the last element of the vector, and `front()` returns the first. (Don't call these functions if the `vector` is empty; that yields undefined behavior.) If you want to refer to a specific element, use `at(n)`, where `n` is a zero-based index, as shown on line 20. The `size()` function (line 19) returns the number of elements in the vector. Therefore, valid indices range from `0` to `size() - 1`.

When you read C++ programs, you will most likely see square brackets (`data[n]`) used to access elements of a vector. The difference between square brackets and the `at` function is that the `at` function provides an additional level of safety. If the index is out of bounds, the program will terminate cleanly. On the other hand, using square brackets with an invalid index will result in undefined behavior: you don't know what will happen. Most dangerous is that your program will not terminate but will continue to run with the bad data. That's why I recommend using `at` for now.

As you can tell from the `std::` prefix, the `vector` type is part of the standard library and is not built into the compiler. Therefore, you need `#include <vector>`, as shown on line 3. No surprises there.

All the functions mentioned so far are member functions; that is, you must supply a `vector` object on the left-hand side of the dot operator (`.`) and the function call on the right-hand side. Another kind of function does not use the dot operator and is free from any particular object. In most languages, this is the typical kind of function, but sometimes C++ programmers call them *free* functions, to distinguish them from member functions. Line 16 shows an example of a free function, `std::sort`.

How would you define a vector of strings?

____

Substitute `std::string` for `int` to get `std::vector<std::string>`. You can also define a `vector` of `vectors`, which is a kind of two-dimensional array: `std::vector<std::vector<int>>`. (Prior to C++ 11, you needed to insert a space between the closing angle brackets, so you will see a lot of code written that way. But in C++ 11, you don't need to concern yourself with that syntax oddity.)

Iterators

The `std::sort` function sorts stuff, as you can tell from the name. In some other object-oriented language, you might expect `vector` to have a `sort()` member function. Alternatively, the standard library could have a `sort` function that can sort anything the library can throw at it. The C++ library falls into the latter category, but with a twist.

The twist is that the `std::sort` function does not take a vector as an argument to sort the contents of the vector. Instead, you pass two values to the `sort` function: the start of a range and the one past the end of the range. The `sort` function sorts the values in that range. The elements in the range can be any type, as long as the compiler knows how to compare objects with the less-than (`<`) operator. The example program specifies the starting position by calling `data.begin()` and one past the end by calling `data.end()`.

These "positions" are called *iterators*. An iterator is an object that can refer to an element of a sequence. As the name implies, an iterator can also iterate over a sequence. The sequence might be elements of a vector or they could be data in a file or database or nodes in a tree. The implementation of the sequence is irrelevant, and the `std::sort` function knows nothing about it. Instead, the `sort` function sees only the iterators.

Iterators present a simple interface, even if their implementation is complicated. The `*` operator returns the value to which the iterator refers (`*iter`), and you can advance an iterator to the next element of the sequence (`++iter`). You can compare two iterators to see if they refer to the same element (`iter1 == iter2`). Iterators come in different flavors, and some flavors let you modify the element or move backward in the sequence.

A bounded loop requires a starting and ending position. One way to specify these for a vector is to specify the positions of the first and last elements, but that raises a thorny issue of what to do with an empty vector. Long ago, computer scientists invented the concept of a sentry or guard. Sometimes, the sentry was a real element added after the last element of a container, marking the position one past the last element. In that case, a container with only the sentry element was empty. Iterators work similarly, but whether the container has a true sentry is a hidden implementation detail. A special iterator value represents the sentry at a position of one past the last element in the container, even if the container has no actual sentry element. The notion of "one past the end" is a common idiom in the C++ library and programs.

Thus, `data.begin()` returns an iterator that refers to the first element of data, and `data.end()` returns an iterator with the special one-past-the-end value, as shown in Figure 9-1.

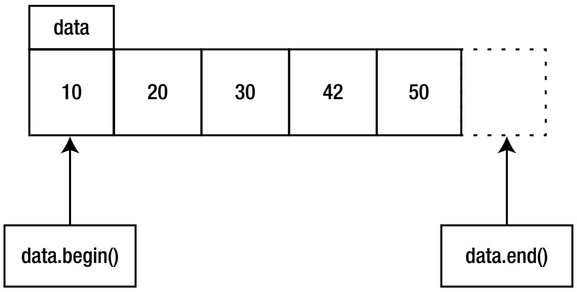


Figure 9-1. *Iterators pointing to positions in a vector*

What is the value of data.begin() if data.size() is zero?

____

That's right. If the vector is empty, `data.begin()` returns the same value as `data.end()`, and that value is a special sentry value that you are not allowed to dereference. In other words, `*data.end()` results in undefined behavior. Because you can compare two iterators, one way to determine if a vector is empty is to test, as demonstrated in the following code:

```
data.begin() == data.end()
```

A better way, however, is to call `data.empty()`, which returns `true` if the vector is empty and `false` if the vector contains at least one element.

Iterators have many uses beyond accessing elements of a vector, and you will see them used often in this book, for input, output, and more, starting with the next Exploration.

EXPLORATION 10

Algorithms and Iterators

The previous Exploration introduced vectors and iterators using `std::sort` to sort a vector of integers. This Exploration examines iterators in more depth and introduces generic algorithms, which perform useful operations on iterators.

Algorithms

The `std::sort` function is an example of a *generic algorithm*, so named because these functions implement common algorithms and operate generically. That is, they work for just about anything you can express as a sequence. Most of the standard algorithms are declared in the `<algorithm>` header, although the `<numeric>` header contains a few that are numerically oriented.

The standard algorithms run the gamut of common programming activities: sorting, searching, copying, comparing, modifying, and more. Searches can be linear or binary. A number of functions, including `std::sort`, reorder elements within a sequence. No matter what they do, all generic algorithms share some common features.

Almost all the generic algorithms operate on iterators. (The sole exceptions are `std::max`, `std::min`, and `std::minmax`, which return the maximum and minimum of two values.) Earlier, I mentioned that iterators come in different flavors, each flavor having different capabilities. Although C++ has five flavors in all, you can broadly group them into two categories: read and write.

A *read* iterator refers to a position in a sequence of values that enables reading from the sequence. The algorithms use read iterators for input but do not modify the values. Typically, you must specify a range with a pair of read iterators: start and one past the end.

A *write* iterator refers to a position in a sequence where the algorithm is to write its output. Typically, you specify only the starting position of the output sequence. The algorithm does not and cannot check for overflow, so you must ensure the output sequence has sufficient room to accommodate everything the algorithm will write.

For example, the `std::copy` algorithm copies values from an input sequence to an output sequence. The function takes three arguments: two read iterators to specify the input range and one write iterator to specify the start of the output range. You must ensure the output has enough capacity. Call the `resize` member function to set the size of the output vector, as shown in Listing 10-1.

Listing 10-1. Demonstrating the `std::copy` Function

```
#include <algorithm>
#include <cassert>
#include <vector>

int main()
{
  std::vector<int> input{ 10, 20, 30 };
  std::vector<int> output{};
  output.resize(input.size());
  std::copy(input.begin(), input.end(), output.begin());
  // Now output has a complete copy of input.
  assert(input == output);
}
```

The `assert` function is a quick way to verify that what you think is true actually is true. You assert a logical statement, and if you are wrong, the program terminates with a message that identifies the assertion. The `assert` function is declared in `<cassert>`; the `c` means the C++ library inherits this header from the C standard library. Note that `assert` is one of the rare exceptions to the rule that standard library members begin with `std::`.

If the program is correct, it runs and exits normally. But if we make a mistake, the assertion triggers, and the programs fails with a message.

Test the program in Listing 10-1. Just to see what happens when an assertion fails, **comment out the call to std::copy and run it again. Write down the message you get.**

____

____

Also note the initialization of `input`. Listing 10-1 demonstrates another application of "universal initialization" (as introduced in Exploration 4). The comma-separated values inside the curly braces are used to initialize the elements of the vector.

Member Types

Remember Listing 9-1? Go back and look at line 19. The definition in the first part of the `for` loop is particularly scary, even for experienced C++ programmers. In addition to member functions, a C++ library type can have member types. In this case, the parent type, `std::vector<int>`, has a member type named `size_type`. Use this type whenever you have to store a size or index for a vector.

You usually use the dot (`.`) operator to call a member function, but you use `::` (called the *scope* operator) to refer to a member type.

The `size_type` is like an `int`, but not really. In particular, you cannot assign a negative value to `size_type`. (After all, what kind of vector has a negative size?) Or rather, the language rules let you assign a negative value, but you won't get the result you want or expect. A good compiler warns you that you are making a mistake. Until you learn enough C++ to appreciate the subtleties of `size_type`, the best strategy is to use `size_type` only for loop control, for storing the size of a vector, and for storing indices. Don't try to perform arithmetic with `size_type` values beyond simply incrementing a loop control variable.

Line 19 uses `size_type` to define the variables `i` and `end`, which are the loop control variables. The loop increments `i` from 0 up to the vector size (`end`), at which point it exits. This is a common idiom for looping through a vector when you need the vector indices. Think of `end` as the index of one past the end, just like iterators. Most programs, in fact, do not need to use the vector indices. I wrote Listing 9-1 that way only to demonstrate the `at` member function.

A better way to write the program in Listing 9-1 is to use iterators instead of indices and the `at()` member function. To define an iterator, substitute the member type `iterator` for `size_type` in the definition of the loop control variable. Initialize the loop control variable to `data.begin()`, and end the loop when the iterator equals `data.end()`. Use the ++ operator to advance the iterator and * to obtain the vector element to which the iterator refers. Put these pieces together and **rewrite lines 19 and 20 of Listing 9-1 to use iterators.**

____

____

Compare your response to Listing 10-2. (For your convenience, I repeat the entire program. That makes it easy for you to compile and test the program.)

Listing 10-2. Sorting Integers by Using Iterators to Print the Results

```
 1 #include <algorithm>
 2 #include <iostream>
 3 #include <vector>
 4
 5 int main()
 6 {
 7   std::vector<int> data{};     // initialized to be empty
 8   int x{};
 9
10   // Read integers one at a time.
11   while (std::cin >> x)
12     // Store each integer in the vector.
13     data.push_back(x);
14
15   // Sort the vector.
16   std::sort(data.begin(), data.end());
17
18   // Print the vector, one number per line.
19   for (std::vector<int>::iterator i{data.begin()}, end{data.end()}; i != end; ++i)
20     std::cout << *i << '\n';
21 }
```

Using iterators instead of indices has many advantages.

- The code works with other kinds of containers (such as linked lists), even if they don't have the `at` member function.
- The optimizer has more opportunities to produce high-performance code.
- You have fewer chances for mistakes, such as buffer overruns.
- The code is easier to read, especially for experienced C++ programmers.

Sure, iterators offer many advantages, but line 19 is a nightmare in terms of readability. The next section introduces a way to hide the nasty iterator syntax, while retaining the advantages of iterators.

A Simpler Loop

C++ 11 introduced an easy way to write a `for` loop. It uses iterators under the hood but hides them in a simple syntax.

```
for (int element : data)
   std::cout << element << '\n';
```

This loop defines a hidden iterator that traverses all the elements of `data` (calling its `begin()` and `end()` member functions). Each time through the loop, the iterator is dereferenced (with the * operator) and assigned to `element`. Thus, inside the loop body, you simply use `element` to obtain each value in the range. After the loop body finishes, the iterator is advanced with the ++ operator.

This style of loop is called a *range-based* `for` loop or a *for-each* loop. Because range-based for loops are new in C++, you will still see a lot of iterators in for loops, but as more people pick up C++, expect to see many more uses of range-based for loops. You can see why.

Using Iterators and Algorithms

Loops over iterator ranges are so common that many generic algorithms implement the most common actions that you may need to take in a program. With a couple of helpers, you can re-implement the program using only generic algorithms, as shown in Listing 10-3.

Listing 10-3. Sorting Integers by Using Only Generic Algorithms and Iterator Adapters

```
 1 #include <algorithm>
 2 #include <iostream>
 3 #include <iterator>
 4 #include <vector>
 5
 6 int main()
 7 {
 8   std::vector<int> data{};
 9
10   // Read integers one at a time.
11   std::copy(std::istream_iterator<int>(std::cin),
12             std::istream_iterator<int>(),
13             std::back_inserter(data));
14
15   // Sort the vector.
16   std::sort(data.begin(), data.end());
17
28   // Print the vector, one number per line.
19   std::copy(data.begin(), data.end(),
20             std::ostream_iterator<int>(std::cout, "\n"));
21 }
```

A `std::istream_iterator` (line 11) creates a read iterator that reads from an input stream. Every time you read a value from the iterator, the `istream_iterator` object uses the `>>` operator to read a value from the stream. You must supply the type in angle brackets, so that the compiler knows what type of information you want to read from the stream, which you pass as a function argument. With no argument, `std::istream_iterator<int>()` (line 12) returns a special one-past-the-end iterator. When the input stream iterator equals this special one-past-the-end iterator, the program has reached the end of the stream, and no more input is available.

The `std::back_inserter` function (line 13) takes a `vector` (or any object that has a `push_back` function) and wraps it in a write iterator. Any time you assign a value to the iterator, the back insert iterator calls the `push_back` function to add the value to the object. Using `back_inserter`, you can guarantee that the program will not overrun the output buffer.

Finally, an `ostream_iterator` (line 20) is the counterpart to an `istream_iterator`. It takes an output stream and wraps it in a write iterator. Any value that you assign to the iterator is written to the stream using the `<<` operator. You can pass an optional string argument, and the `ostream_iterator` writes that string after each value. In this case, the string contains just a newline character, so each number is written on its own line.

All these special iterators are declared in `<iterator>`. You don't need this header to use an ordinary iterator, such as that returned from a `vector`'s `begin()` function, but you do need it if you use a special iterator, such as `istream_iterator`.

Until you are accustomed to using the generic algorithms, iterators, and special iterators, this style of programming can seem unusual. Once you are familiar with these unique C++ library members, you will find such code easier to read and write than more traditional programming styles.

It's now time for you to practice using iterators and algorithms.

Write a program that reads integers from the standard input into a vector. Feel free to use any style of loop (indices, iterators, or range-based). If you are unsure which style to use, I recommend getting used to range-based loops. Print each value, followed by twice the value and then the value squared. Thus, the output contains one line per input value, and each line has three numbers. Align the columns, as you learned in the previous Exploration.

Test your program using the following input:

```
3
10
8
```

What output do you expect?

____

____

____

What output do you actually get?

____

____

____

Now try running the program with no input at all. **What do you expect?**

____

What do you get?

____

Listing 10-4 shows one way to write this program using explicit loops. Notice how the * operator means "multiplication" when used as a binary (two-operand) operator and "dereference the iterator" when used as a unary, prefix operator.

Listing 10-4. Doubling and Squaring Input Values in a Vector by Using Iterators

```
#include <iomanip>
#include <iostream>
#include <vector>

int main()
{
   std::vector<int> data{};
   int x{};

   while (std::cin >> x)
      data.push_back(x);
   for (std::vector<int>::iterator i{data.begin()}, end{data.end()}; i != end; ++i)
      std::cout << std::setw(2) << *i <<
         std::setw(3) << *i* 2 <<
         std::setw(4) << *i * *i << '\n';
}
```

The expression `*i * *i` is hard to read. Listing 10-5 shows another way, this time using a range-based for loop. See how much easier it is to read.

Listing 10-5. Doubling and Squaring Input Values in a Vector by Using a Range-Based Loop

```
#include <iomanip>
#include <iostream>
#include <vector>

int main()
{
   std::vector<int> data{};
   int x{};

   while (std::cin >> x)
      data.push_back(x);

   std::cout.fill(' ');
   for (int element : data)
      std::cout << std::setw(2) << element <<
         std::setw(3) << element * 2 <<
         std::setw(4) << element * element << '\n';
}
```

The ++ operator, which advances an iterator, also advances an integer, as shown in Listing 10-4. The next Exploration takes a closer look at this handy operator.

EXPLORATION 11

Increment and Decrement

The previous Exploration introduced the increment (++) operator to advance an iterator. This operator works on numeric types as well. Not surprisingly, it has a decrement counterpart: `--`. This Exploration takes a closer look at these operators, which appear so often, they are part of the language name.

Note I know that you C, Java, etc. programmers have been waiting for this Exploration ever since I wrote `i = i + 1` in Exploration 7. As you saw in the previous Exploration, the ++ operator means more in C++ than what you're used to. That's why I waited until now to introduce it.

Increment

The ++ operator is familiar to C, Java, Perl, and many other programmers. C was the first widespread language to introduce this operator to mean "increment" or "add 1." C++ expanded the usage it inherited from C; the standard library uses the ++ operator in several new ways, such as advancing an iterator (as you saw in the previous Exploration).

The increment operator comes in two flavors: prefix and postfix. The best way to understand the difference between these two flavors is with a demonstration, as shown in Listing 11-1.

Listing 11-1. Demonstrating the Difference Between Prefix and Postfix Increment

```
#include <iostream>

int main()
{
  int x{42};

  std::cout << "x   = " << x   << "\n";
  std::cout << "++x = " << ++x << "\n";
  std::cout << "x   = " << x   << "\n";
  std::cout << "x++ = " << x++ << "\n";
  std::cout << "x   = " << x   << "\n";
}
```

Predict the output of the program.

____

____

____

____

____

What is the actual output?

____

____

____

____

____

Explain the difference between prefix (`++x`) and postfix (`x++`) increment.

____

____

____

Described briefly, the prefix operator increments the variable first: the value of the expression is the value after incrementing. The postfix operator saves the old value, increments the variable, and uses the old value as the value of the expression.

As a general rule, use prefix instead of postfix, unless you need the postfix functionality. Rarely is the difference significant, but the postfix operator must save a copy of the old value, which might impose a small performance cost. If you don't have to use postfix, why pay that price?

Decrement

The increment operator has a decrement counterpart: `--`. The decrement operator subtracts one instead of adding one. Decrement also has a prefix and postfix flavor. The prefix operator pre-decrements, and the postfix operator post-decrements.

You can increment and decrement any variable with a numeric type; however, only some iterators permit decrement.

For example, write iterators move forward only. You can use the increment operator (prefix or postfix), but not decrement. Test this for yourself. Write a program that uses `std::ostream_iterator` and try to use the decrement operator on the iterator. (If you need a hint, look at Listing 10-4. Save the `ostream_iterator` object in a variable. Then use the decrement operator. It doesn't matter that the program makes no sense; it won't get past the compiler anyway.)

What error message do you get?

____

____

Different compilers issue different messages, but the essence of the message should be that the `--` operator is not defined. If you need help with the program, see Listing 11-2.

Listing 11-2. Erroneous Program That Applies Decrement to an Output Iterator

```
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
  std::vector<int> data{ 10, 42 };
  std::ostream_iterator<int> out{ std::ostream_iterator<int>(std::cout, "") };
  std::copy(data.begin(), data.end(), out);
  --out;

}
```

A `vector`'s iterators allow increment and decrement. Using increment and decrement operators on iterators, **write a program that reads integers from the standard input into a vector, reverses the order of the vector, and writes the result.** (No fair peeking in a language reference and using the `std::reverse` algorithm. Use two iterators: one pointing to the start of the vector and the other pointing to the end. Stop the loop when the iterators meet. Make sure they don't pass each other, and make sure your program does not try to dereference the one-past-the-end iterator.)

Test your program on input with both an even and an odd number of integers. Compare your program with the one in Listing 11-3.

Listing 11-3. Reversing the Input Order

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
  std::vector<int> data{};
  int x{};
  while (std::cin >> x)
    data.push_back(x);

  for (std::vector<int>::iterator start{data.begin()}, end{data.end()};
       start != end;
       /*empty*/)
  {
    --end;
    if (start != end)
    {
      std::iter_swap(start, end);
      ++start;
    }
  }

  std::copy(data.begin(), data.end(), std::ostream_iterator<int>(std::cout, "\n"));
}
```

The start iterator points to the beginning of the data vector, and end initially points to one past the end. If the vector is empty, the for loop terminates without executing the loop body. Then the loop body decrements end so that it points to an actual element of the vector. If the vector contains an even number of elements, the if condition is true, so std::iter_swap is called, and the program advances start one position.

As the name implies, std::iter_swap swaps the values stored at two iterators, in this case, the iterators start and end. You can think of it as performing the following:

```
int tmp = *start;
*start = *end;
*end = tmp;
```

Notice that the program is careful to compare start != end after each increment or decrement operation. If the program had only one comparison, it would be possible for start and end to pass each other. The loop condition would never be true, and the program would exhibit undefined behavior, so the sky would fall, the earth would swallow me, or worse.

Also note how the for loop has an empty postiteration part. The iteration logic appears in different places in the loop body, which is not the preferred way to write a loop but is necessary in this case.

You can rewrite the loop, so the postiteration logic appears only in the loop header. Some programmers argue that distributing the increment and decrement in the loop body makes the loop harder to understand and, in particular, harder to prove the loop terminates correctly. On the other hand, cramming everything in the loop header makes the loop condition especially tricky to understand, as you can see in Listing 11-4.

Listing 11-4. Rewriting the for Loop

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
  std::vector<int> data{};
  int x{};
  while (std::cin >> x)
    data.push_back(x);

  for (std::vector<int>::iterator start{data.begin()}, end{data.end()};
       start != end and start != --end;
       ++start)
  {
      std::iter_swap(start, end);
  }

  std::copy(data.begin(), data.end(), std::ostream_iterator<int>(std::cout, "\n"));
}
```

To keep all the logic in the loop header, it was necessary to use a new operator: and. You will learn more about this operator in the next Exploration; meanwhile, just believe that it implements a logical and operation and keep reading.

Most experienced C++ programmers will probably prefer Listing 11-4, whereas most beginners will probably prefer Listing 11-3. Hiding a decrement in the middle of a condition makes the code harder to read and understand. It's too easy to overlook the decrement. As you gain experience with C++, however, you will become more comfortable with increments and decrements, and Listing 11-4 will start to grow on you.

■ **Note** I prefer Listing 11-3 over Listing 11-4. I really don't like to bury increment and decrement operators in the middle of a complicated condition.

So what else would experienced C++ programmers do? Because they have broader knowledge of the C++ standard library, they would make better use of it. In particular, they would use the `std::reverse` algorithm, which reverses the elements in a range.

```
std::reverse(data.begin(), data.end());
```

Another idea is to use `istream_iterator`, which you learned about in Exploration 9. This time, you will use it a little differently. Instead of using `back_inserter` (introduced in the previous Exploration) and the `copy` algorithm, Listing 11-5 calls the `insert` member function, which copies values from any iterator range into the vector. The values are inserted before the position given by the first argument (`data.end()`, in this case).

Listing 11-5. Taking Advantage of the Standard Library

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
  std::vector<int> data{};

  // Read integers from the standard input, and append them to the end of data.
  data.insert(data.end(),
              std::istream_iterator<int>(std::cin), std::istream_iterator<int>());

  // Reverse the order of the contents of data.
  std::reverse(data.begin(), data.end());

 // Print data, one number per line.
  std::copy(data.begin(), data.end(), std::ostream_iterator<int>(std::cout, "\n"));
}
```

As you learn more C++, you will find other aspects of this program that lend themselves to improvement. I encourage you to revisit old programs and see how your new techniques can often simplify the programming task. I'll do the same as I revisit examples throughout this book.

Listing 11-4 introduced the `and` operator. The next Exploration takes a closer look at this operator, as well as other logical operators and their use in conditions.

EXPLORATION 12

Conditions and Logic

You first met the bool type in Exploration 2. This type has two possible values: true and false, which are reserved keywords (unlike in C). Although most Explorations have not needed to use the bool type, many have used logical expressions in loop and if-statement conditions. This Exploration examines the many aspects of the bool type and logical operators.

I/O and bool

C++ I/O streams permit reading and writing bool values. By default, streams treat them as numeric values: true is 1 and false is 0. The manipulator std::boolalpha (declared in <ios>, so you get it for free from <iostream>) tells a stream to interpret bool values as words. By default, the words are true and false. (In Exploration 18, you'll discover how to use a language other than English.) You use the std::boolalpha manipulator the same way you do any other manipulator (as you saw in Exploration 8). For an input stream, use an input operator with the manipulator.

Write a program that demonstrates how C++ formats and prints bool values, numerically and textually. Compare your program with Listing 12-1.

Listing 12-1. Printing bool Values

```
#include <iostream>

int main()
{
  std::cout << "true=" << true << '\n';
  std::cout << "false=" << false << '\n';
  std::cout << std::boolalpha;
  std::cout << "true=" << true << '\n';
  std::cout << "false=" << false << '\n';
}
```

How do you think C++ handles bool values for input?

____

Write a program to test your assumptions. Were you correct? _________________ Explain how an input stream handles bool input:

____

____

By default, when an input stream has to read a bool value, it actually reads an integer, and if the integer's value is 1, the stream interprets that as true. The value 0 is false, and any other value results in an error.

With the std::boolalpha manipulator, the input stream requires the exact text true or false. Integers are not allowed, nor are any case differences. The input stream accepts only those exact words.

Use the std::noboolalpha manipulator to revert to the default numeric Boolean values. Thus, you can mix alphabetic and numeric representations of bool in a single stream, as follows:

```
bool a{true}, b{false};
std::cin >> std::boolalpha >> a >> std::noboolalpha >> b;
std::cout << std::boolalpha << a << ' ' << std::noboolalpha << b;
```

Reading and writing bool values does not actually occur all that often in most programs.

Boolean Type

C++ automatically converts many different types to bool, so you can use integers, I/O stream objects, and other values whenever you need a bool, such as in a loop or if-statement condition. You can see this for yourself in Listing 12-2.

Listing 12-2. Automatic Type Conversion to bool

```
#include <iostream>

int main()
{
  if (true)         std::cout << "true\n";
  if (false)        std::cout << "false\n";
  if (42)           std::cout << "42\n";
  if (0)            std::cout << "0\n";
  if (42.4242)      std::cout << "42.4242\n";
  if (0.0)          std::cout << "0.0\n";
  if (-0.0)         std::cout << "-0.0\n";
  if (-1)           std::cout << "-1\n";
  if ('\0')         std::cout << "'\\0'\n";
  if ('\1')         std::cout << "'\\1'\n";
  if ("1")          std::cout << "\"1\"\n";
  if ("false")      std::cout << "\"false\"\n";
  if (std::cout)    std::cout << "std::cout\n";
  if (std::cin)     std::cout << "std::cin\n";
}
```

Predict the output from Listing 12-2:

____

____

____

____

Check your answer. **Were you right?** ________________

You may have been fooled by lines 15 and 16. C++ does not interpret the contents of string literals to decide whether to convert the string to `true` or `false`. All character strings are `true`, even empty ones. (The C++ language designers did not do this to be perverse. There's a good reason that strings are `true`, but you will have to learn quite a bit more C++ in order to understand why.)

On the other hand, character literals (lines 13 and 14) are completely different from string literals. The compiler converts the escape character `'\0'`, which has numeric value zero, to `false`. All other characters are `true`.

Recall from many previous examples (especially in Exploration 3) that loop conditions often depend on an input operation. If the input succeeds, the loop condition is `true`. What is actually happening is that C++ knows how to convert a stream object (such as `std::cin`) to `bool`. Every I/O stream keeps track of its internal state, and if any operation fails, the stream remembers that fact. When you convert a stream to `bool`, if the stream is in a failure state, the result is `false`. Not all complex types can be converted to `bool`, however.

What do you expect to happen when you compile and run Listing 12-3?

____

Listing 12-3. Converting a `std::string` to `bool`

```
#include <iostream>
#include <string>

int main()
{
  std::string empty{};

  if (empty)
    std::cout << "empty is true\n";
  else
    std::cout << "empty is false\n";

}
```

The compiler reports an error, because it does not know how to convert `std::string` to `bool`.

■ **Note** Although an `istream` knows how to convert an input string to `bool`, the `std::string` type lacks the information it needs to interpret the string. Without knowledge of the string's context, it is unrealistic to ask a string to interpret text, such as "true," "vrai," or "richtig."

What about `std::vector`? **Do you think C++ defines a conversion from std::vector to bool? ________________. Write a program to test your hypothesis. What is your conclusion?**

____

This is another case in which no general solution is feasible. Should an empty vector be `false`, whereas all others are `true`? Maybe a `std::vector<bool>` that contains only `false` elements should be `false`. Only the application programmer can make these decisions, so the C++ library designers wisely chose not to make them for you; therefore, you cannot convert `std::vector` to `bool`. However, there are ways of obtaining the desired result by calling member functions.

Logic Operators

Real-world conditions are often more complicated than merely converting a single value to `bool`. To accommodate this, C++ offers the usual logical operators: `and`, `or`, and `not` (which are reserved keywords). They have their usual meaning from mathematical logic, namely that `and` is `false`, unless both operands are `true`, `or` is `true`, unless both operands are `false`, and `not` inverts the value of its operand.

More important, however, is that the built-in `and` and `or` operators do not evaluate their right-hand operand, unless they have to. The `and` operator must evaluate its right-hand operand, only if the left-hand operand is `true`. (If the left-hand operand is `false`, the entire expression is `false`, and there is no need to evaluate the right-hand operand.) Similarly, the `or` operator evaluates its right-hand operand only if the left-hand operand is `true`. Stopping the evaluation early like this is known as *short-circuiting*.

For example, suppose you are writing a simple loop to examine all the elements of a vector to determine whether they are all equal to zero. The loop ends when you reach the end of the vector or when you find an element not equal to zero.

Write a program that reads numbers into a vector, searches the vector for a nonzero element, and prints a message about whether the vector is all zero.

You can solve this problem without using a logical operator, but try to use one, just for practice. Take a look at Listing 12-4, to see one way to solve this problem.

Listing 12-4. Using Short-Circuiting to Test for Nonzero Vector Elements

```
 1 #include <iostream>
 2 #include <iterator>
 3 #include <vector>
 4
 5 int main()
 6 {
 7   std::vector<int> data{};
 8   data.insert(data.begin(),
 9               std::istream_iterator<int>(std::cin),
10               std::istream_iterator<int>());
11
12   std::vector<int>::iterator iter{}, end{data.end()};
13   for (iter = data.begin(); iter != end and *iter == 0; ++iter)
14     /*empty*/;
15   if (iter == data.end())
16     std::cout << "data contains all zeroes\n";
17   else
18     std::cout << "data does not contain all zeroes\n";
19 }
```

Line 13 is the key. The iterator advances over the vector and tests for zero-valued elements.

What happens when the iterator reaches the end of the vector?

____

The condition `iter != data.end()` becomes `false` at the end of the vector. Because of short-circuiting, C++ never evaluates the `*iter == 0` part of the expression, which is good.

Why is this good? What would happen if short-circuiting did not take place?

____

____

Imagine that `iter != end` is `false`; in other words, the value of `iter` is `end`. That means `*iter` is just like `*end`, which is bad—really bad. You are not allowed to dereference the one-past-the-end iterator. If you are lucky, it would crash your program. If you are unlucky, your program would continue to run, but with completely unpredictable and erroneous data, and, therefore, unpredictable and erroneous results.

Short-circuiting guarantees that C++ will not evaluate `*iter` when `iter` equals `end`, which means `iter` will always be valid when the program dereferences it, which is good. Some languages (such as Ada) use different operators for short-circuiting and non–short-circuiting operations. C++ does not. The built-in logical operators always perform short-circuiting, so you never accidentally use non–short-circuiting when you intended to use the short-circuiting operator.

Old-Fashioned Syntax

The logical operators have symbolic versions: `&&` for `and`, `||` for `or`, and `!` for `not`. The keywords are clearer, easier to read, easier to understand, and less error-prone. That's right, less error-prone. You see, `&&` means `and`, but `&` is also an operator. Similarly, `|` is a valid operator. Thus, if you accidentally write `&` instead of `&&`, your program will compile and even run. It might seem to run correctly for a while, but it will eventually fail, because `&` and `&&` mean different things. (You'll learn about `&` and `|` later in this book.) New C++ programmers aren't the only ones to make this mistake. I've seen highly experienced C++ programmers write `&` when they mean `&&`, or `|` instead of `||`. Avoid this error by using only the keyword logical operators.

I was hesitant about even mentioning the symbolic operators, but I can't ignore them. Many C++ programs use the symbolic operators instead of the keyword equivalents. These C++ programmers, having grown up with the symbols, prefer to continue to use the symbols rather than the keywords. This is your chance to become a trendsetter. Eschew the old-fashioned, harder-to-read, harder-to-understand, error-prone symbols and embrace the keywords.

Comparison Operators

The built-in comparison operators always yield `bool` results, regardless of their operands. You have already seen `==` and `!=` for equality and inequality. You also saw `<` for less than, and you can guess that `>` means greater than. Similarly, you probably already know that `<=` means less than or equal and `>=` means greater than or equal.

These operators produce the expected results when you use them with numeric operands. You can even use them with vectors of numeric types.

Write a program that demonstrates how < works with a vector of int. (If you're having trouble writing the program, take a look at Listing 12-5.) **What are the rules that govern < for a vector?**

____

____

____

____

Listing 12-5. Comparing Vectors

```
#include <iostream>
#include <vector>

int main()
{
   std::vector<int> a{ 10, 20, 30 },  b{ 10, 20, 30 };

   if (a != b) std::cout << "wrong: a != b\n";
   if (a < b)  std::cout << "wrong: a < b\n";
   if (a > b)  std::cout << "wrong: a > b\n";
   if (a == b) std::cout << "okay: a == b\n";
   if (a >= b) std::cout << "okay: a >= b\n";
   if (a <= b) std::cout << "okay: a <= b\n";

   a.push_back(40);
   if (a != b) std::cout << "okay: a != b\n";
   if (a < b)  std::cout << "wrong: a < b\n";
   if (a > b)  std::cout << "okay: a > b\n";
   if (a == b) std::cout << "wrong: a == b\n";
   if (a >= b) std::cout << "okay: a >= b\n";
   if (a <= b) std::cout << "wrong: a <= b\n";

   b.push_back(42);
   if (a != b) std::cout << "okay: a != b\n";
   if (a < b)  std::cout << "okay: a < b\n";
   if (a > b)  std::cout << "wrong: a > b\n";
   if (a == b) std::cout << "wrong: a == b\n";
   if (a >= b) std::cout << "wrong: a >= b\n";
   if (a <= b) std::cout << "okay: a <= b\n";
}
```

C++ compares vectors at the element level. That is, the first elements of two vectors are compared. If one element is smaller than the other, its vector is considered less than the other. If one vector is a prefix of the other (that is, the vectors are identical up to the length of the shorter vector), the shorter vector is less than the longer one.

C++ uses the same rules when comparing `std::string` types, but not when comparing two character string literals.

Write a program that demonstrates how C++ compares two `std::string` objects by comparing their contents.

Compare your solution with mine in Listing 12-6.

Listing 12-6. Demonstrating How C++ Compares Strings

```
#include <iostream>
#include <string>

int main()
{
   std::string a{"abc"}, b{"abc"};
   if (a != b) std::cout << "wrong: abc != abc\n";
   if (a < b)  std::cout << "wrong: abc < abc\n";
   if (a > b)  std::cout << "wrong: abc > abc\n";
   if (a == b) std::cout << "okay: abc == abc\n";
   if (a >= b) std::cout << "okay: abc >= abc\n";
   if (a <= b) std::cout << "okay: abc <= abc\n";

   a.push_back('d');
   if (a != b) std::cout << "okay: abcd != abc\n";
   if (a < b)  std::cout << "wrong: abcd < abc\n";
   if (a > b)  std::cout << "okay: abcd > abc\n";
   if (a == b) std::cout << "wrong: abcd == abc\n";
   if (a >= b) std::cout << "okay: abcd >= abc\n";
   if (a <= b) std::cout << "wrong: abcd <= abc\n";

   b.push_back('e');
   if (a != b) std::cout << "okay: abcd != abce\n";
   if (a < b)  std::cout << "okay: abcd < abce\n";
   if (a > b)  std::cout << "wrong: abcd > abce\n";
   if (a == b) std::cout << "wrong: abcd == abce\n";
   if (a >= b) std::cout << "wrong: abcd >= abce\n";
   if (a <= b) std::cout << "okay: abcd <= abce\n";
}
```

Testing how C++ compares quoted string literals is more difficult. Instead of using the contents of the string, the compiler uses the location of the strings in memory, which is a detail of the compiler's internal workings and has no bearing on anything practical. Thus, unless you know how the compiler works, you cannot predict how it will compare two quoted strings. In other words, don't do that. Make sure you create `std::string` objects before you compare strings. It's okay if only one operand is `std::string`. The other can be a quoted string literal, and the compiler knows how to compare the `std::string` with the literal, as demonstrated in the following example:

```
if ("help" > "hello") std::cout << "Bad. Bad. Bad. Don't do this!\n";
if (std::string("help") > "hello") std::cout << "this works\n";
if ("help" > std::string("hello")) std::cout << "this also works\n";
if (std::string("help") > std::string("hello")) std::cout << "and this works\n";
```

The next Exploration does not relate directly to Boolean logic and conditions. Instead, it shows how to write compound statements, which you need in order to write any kind of useful conditional statement.

EXPLORATION 13

Compound Statements

You have already used compound statements (that is, lists of statements enclosed in curly braces) in many programs. Now it's time to learn some of the special rules and uses for compound statements, which are also known as *blocks*.

Statements

C++ has some hairy, scary syntax rules. By comparison though, the syntax for statements is downright simplistic. The C++ grammar defines most statements in terms of other statements. For example, the rule for `while` statements is

`while (` *`condition`* **`)`** *`statement`*

In this example, bold elements are required, such as the `while` keyword. *Italic* elements stand for other syntax rules. As you can likely deduce from the example, a `while` statement can have any statement as the loop body, including another `while` statement.

The reason most statements appear to end with a semicolon is because the most fundamental statement in C++ is just an expression followed by a semicolon.

`expression` **`;`**

This kind of statement is called an *expression statement*.

I haven't discussed the precise rules for expressions yet, but they work the way they do in most other languages, with a few differences. Most significant is that assignment is an expression in C++ (as it is in C, Java, C#, etc., but not in Pascal, Basic, Fortran, etc.). Consider the following:

```
while (std::cin >> x)
  sum = sum + x;
```

This example demonstrates a single `while` statement. Part of the `while` statement is another statement: in this case, an expression statement. The expression in the expression statement is `sum = sum + x`. Expressions in expression statements are often assignments or function calls, but the language permits any expression. The following, therefore, is a valid statement:

```
42;
```

What do you think happens if you use this statement in a program?

____

Try it. What actually happens?

____

Modern compilers are usually able to detect statements that serve no useful purpose and eliminate them from the program. Typically, the compiler tells you what it's doing, but you may have to supply an extra option to tell the compiler to be extra picky. For example, try the `-Wall` option for g++ or `/Wall` for Microsoft Visual C++. (That's Wall, as in all warnings, not the thing holding up your roof.)

The syntax rule for a compound statement is

```
{ statement* }
```

where * means zero or more occurrences of the preceding rule (*`statement`*). Notice that the closing curly brace has no semicolon after it.

How does C++ parse the following?

```
while (std::cin >> x)
{
    sum = sum + x;
    ++count;
}
```

____

____

____

Once again, you have a `while` statement, so the loop body must be a single statement. In this example, the loop body is a compound statement. The compound statement is a statement that consists of two expression statements. Figure 13-1 shows a tree view of the same information.

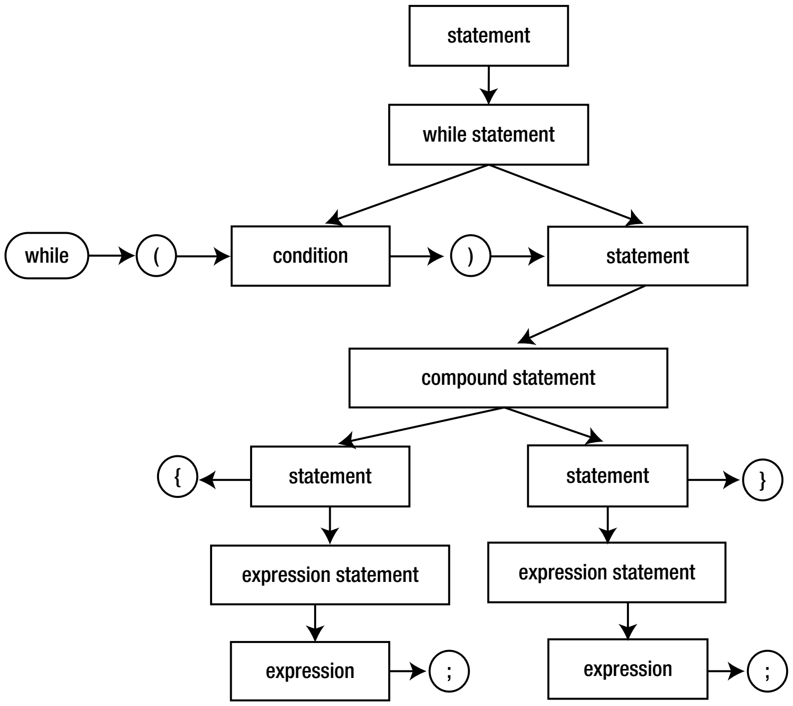


***Figure 13-1.** Simplified parse tree for C++ statements*

Consider the body of `main()`, such as the one in Listing 13-1. What do you see? That's right, it's a compound statement. It's an ordinary block, and it follows the same rules as any other block. In case you were wondering, the body of `main()` must be a compound statement. This is one of the few circumstances in which C++ requires a specific kind of statement, instead of allowing any statement whatsoever.

Find and fix the errors in Listing 13-1. Visually locate as many errors as you can by reading the code. When you think you found and fixed them all, try compiling and running the program.

Listing 13-1. Finding Statement Errors

```
 1 #include <iostream>
 2 #include <vector>
 3 // find errors in this program
 4 int main()
 5 {
 6   std::vector<int> positive_data{}, negative_data{};
 7
 8   for (int x{0}; std::cin >> x ;) {
 9     if (x < 0);
10     {
11       negative_data.push_back(x)
12     };
13     else
14     {
15       positive_data.push_back(x)
16     }
17   };
18 }
```

Record all the errors in Listing 13-1.

Did you find them all without the compiler's help? ________________

The errors are:

- Extra semicolon on line 9
- Extra semicolon on line 12
- Missing semicolon from the end of lines 11 and 15
- Extra semicolon on line 17

For extra credit, **which errors are not syntax violations (the compiler will not alert you to them) and do not affect the program's behavior?**

If you answered "the extra semicolon on line 17," give yourself a star. Strictly speaking, the extra semicolon represents an empty, do-nothing statement, called a *null statement*. Such a statement sometimes has a use in a loop, especially a `for` loop that does all its work in the loop header, leaving nothing for the loop body to do. (See an example in Listing 12-4.)

Thus, the way the compiler interprets line 9 is that the semicolon is the statement body for the `if` statement. The next statement is a compound statement, which is followed by an `else`, which has no corresponding `if`, hence the error. Every `else` must be a counterpart to an earlier `if` in the same statement. In other words, every `if` condition must be followed by exactly one statement, then by an optional `else` keyword and another statement. You cannot use `else` in any other way.

As written, the `if` statement on line 9 is followed by three statements: a null statement, a compound statement, and another null statement. The solution is to delete the null statements by deleting the semicolons on lines 9 and 12.

The statements that make up a compound statement can be any statements, including other compound statements. The next section explains why you might want to nest a compound statement within another compound statement.

Line 6 shows that you can declare more than one variable at a time, using a comma separator. I prefer to define one variable at a time but wanted to show you this style too. Each variable receives its own initializer.

Local Definitions and Scope

Compound statements do more than simply group multiple statements into a single statement. You can also group definitions within the block. Any variable that you define in a block is visible only within the confines of the block. The region where you can use a variable is known as the variable's *scope*. A good programming practice is to limit the scope to as small a region as possible. Limiting the scope of a variable serves several purposes.

- *Preventing mistakes*: You can't accidentally use a variable's name outside of its scope.
- *Communicating intent*: Anyone who reads your code can tell how a variable is used. If variables are defined at the broadest scope possible, whoever reads your code must spend more time and effort trying to determine where different variables are used.
- *Reusing names*: How many times can you use the variable `i` as a loop control variable? You can use and reuse it as often as you like, provided each time you limit the variable's scope to its loop.
- *Reusing memory*: When execution reaches the end of a block, all the variables defined in that block are destroyed, and the memory is available to be used again. Thus, if your code creates many large objects but needs only one at a time, you can define each variable in its own scope, so only one large object exists at a time.

Listing 13-2 shows some examples of local definitions. The lines highlighted in bold indicate local definitions.

Listing 13-2. Local Variable Definitions

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
```

```
int main()
{
  std::vector<int> data{};
  data.insert(data.begin(), std::istream_iterator<int>(std::cin),
                            std::istream_iterator<int>());

  // Silly way to sort a vector. Assume that the initial portion
  // of the vector has already been sorted, up to the iterator iter.
  // Find where *iter belongs in the already sorted portion of the vector.
  // Erase *iter from the vector and re-insert it at its sorted position.
  // Use binary search to speed up the search for the proper position.
  // Invariant: elements in range [begin(), iter are already sorted.
  for (std::vector<int>::iterator iter{data.begin()}, end{data.end()}; iter != end; )
  {
    // Find where *iter belongs by calling the standard algorithm
    // lower_bound, which performs a binary search and returns an iterator
    // that points into data at a position where the value should be inserted.
    int value{*iter};
    std::vector<int>::iterator here{std::lower_bound(data.begin(), iter, value)};
    if (iter == here)
      ++iter; // already in sorted position
    else
    {

      // erase the out-of-position item, advancing iter at the same time.

      iter = data.erase(iter);
      // re-insert the value at the correct position.

      data.insert(here, value);

    }

  }

  // Debugging code: check that the vector is actually sorted. Do this by comparing
  // each element with the preceding element in the vector.
  for (std::vector<int>::iterator iter{data.begin()}, prev{data.end()}, end{data.end()};
       iter != end;
      ++iter)
  {
    if (prev != data.end())
      assert(not (*iter < *prev));
     prev = iter;
  }

  // Print the sorted vector all on one line. Start the line with "{" and
  // end it with "}". Separate elements with commas.
  // An empty vector prints as "{ }".
  std::cout << '{';
  std::string separator{" "};
  for (int element : data)
  {
    std::cout << separator << element;
    separator = ", ";
  }
  std::cout << " }\n";
}
```

Listing 13-2 has a lot of new functions and features, so let's look at the code one section at a time.

The definition of data is a local definition in a block. True, almost all your definitions have been at this outermost level, but a compound statement is a compound statement, and any definition in a compound statement is a local definition. That begs the question of whether you can define a variable outside of all blocks. The answer is yes, but you rarely want to. C++ permits global variables, but no program in this book has needed to define any yet. I'll cover global variables when the time is right (which would be Exploration 52).

A for loop has its own special scope rules. As you learned in Exploration 7, the initialization part of a for loop can, and often does, define a loop control variable. The scope of that variable is limited to the for loop, as though the for statement were enclosed in an extra set of curly braces.

The value variable is also local to the for loop's body. If you try to use this variable outside of the loop, the compiler issues an error message. In this case, you have no reason to use this variable outside the loop, so define the variable inside the loop.

The lower_bound algorithm performs a binary search that tries to find a value in a range of sorted values. It returns an iterator that points to the first occurrence of the value in the range, or, if the value is not found, the position where you can insert the value and keep the range in order. This is exactly what this program needs to sort the data vector.

The erase member function deletes an element from a vector, reducing the vector's size by one. Pass an iterator to erase to designate which element to delete, and save the return value, which is an iterator that refers to the new value at that position in the vector. The insert function inserts a value (the second argument) just before the position designated by an iterator (the first argument).

Notice how you can use and reuse the name iter. Each loop has its own distinct variable named iter. Each iter is local to its loop. If you were to write sloppy code and fail to initialize iter, the variable's initial value would be junk. It is not the same variable as the one defined earlier in the program, so its value is not the same as the old value of the old variable.

The separator variable holds a separator string to print between elements when printing the vector. It, too, is a local variable, but local to the main program's block. However, by defining it just before it is used, you communicate the message that this variable is not needed earlier in main. It helps prevent mistakes that can arise if you reuse a variable from one part of main in another part.

Another way you can help limit the scope of a variable such as separator is to define it in a block within a block, as shown in Listing 13-3. (This version of the program replaces the loops with calls to standard algorithms, which is a better way to write C++ programs when you are not trying to make a point.)

Listing 13-3. Local Variable Definitions in a Nested Block

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
```

```
int main()
{
  std::vector<int> data{};
  data.insert(data.begin(), std::istream_iterator<int>(std::cin),
                            std::istream_iterator<int>());

  std::sort(data.begin(), data.end());

  {
    // Print the sorted vector all on one line. Start the line with "{" and
    // end it with "}". Separate elements with commas. An empty vector prints
    // as "{ }".
    std::cout << '{';
    std::string separator{" "};
    for (int element : data)
    {
      std::cout << separator << element;
      separator = ", ";
    }
    std::cout << " }\n";
  }
  // Cannot use separator out here.
}
```

Most C++ programmers nest blocks infrequently. As you learn more C++, you will discover a variety of techniques that improve on nested blocks and keep your `main` program from looking so cluttered.

Definitions in for Loop Headers

What if you did not define loop control variables inside the `for` loop header, but defined them outside the loop instead? Try it.

Rewrite Listing 13-2, so you don't define any variables in the for loop headers.

What do you think? **Does the new code look better or worse than the original?** __________ **Why?**

____

____

____

Personally, I find `for` loops can become cluttered very easily. Nonetheless, keeping loop control variables local to the loop is critical for clarity and code comprehension. When faced with a large, unknown program, one of the difficulties you face in understanding that program is knowing when and how variables can take on new values. If a variable is local to a loop, you know the variable cannot be modified outside the loop. That is valuable information. If you still need convincing, try reading and understanding Listing 13-4.

Listing 13-4. Mystery Function

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

int main()
{
  int v{};
  std::vector<int> data{};
  std::vector<int>::iterator i{}, p{};
  std::string s{};

  std::copy(std::istream_iterator<int>(std::cin),
            std::istream_iterator<int>(),
            std::back_inserter(data));
  i = data.begin();

  while (i != data.end())
  {
    v = *i;
    p = std::lower_bound(data.begin(), i, v);
    i = data.erase(i);
    data.insert(p, v);
  }

  s = " ";
  for (p = i, i = data.begin(); i != data.end(); p = i, ++i)
  {
    if (p != data.end())
      assert(not (*i < *p));
  }

  std::cout << '{';
  for (i = data.begin(); i != data.end(); ++i)
  {
    v = *p;
    std::cout << s << v;
    s = ", ";
  }
  std::cout << " }\n";
}
```

Well, that wasn't too hard, was it? After all, you recently finished reading Listing 13-2, so you can see that Listing 13-4 is intended to do the same thing but is reorganized slightly. The difficulty is in keeping track of the values of `p` and `i`, and ensuring that they have the correct value at each step of the program. Try compiling and running the program. **Record your observations.**

____

____

____

What went wrong?

____

____

____

I made a mistake and wrote `v = *p` instead of `v = *i`. Congratulations if you spotted this error before you ran the program. If the variables had been properly defined local to their respective scopes, this error could never have occurred.

The next Exploration introduces file I/O, so your exercises can read and write files, instead of using console I/O. I'm sure your fingers will appreciate it.

EXPLORATION 14

Introduction to File I/O

Reading from standard input and writing to standard output works fine for many trivial programs, and it is a standard idiom for UNIX and related operating systems. Nonetheless, real programs must be able to open named files for reading, writing, or both. This Exploration introduces the basics of file I/O. Later Explorations will tackle more sophisticated I/O issues.

Reading Files

The most common file-related task in these early Explorations will involve reading from a file instead of from the standard input stream. One of the greatest benefits of this is it saves a lot of tedious typing. Some IDEs make it difficult to redirect input and output, so it's easier to read from a file and sometimes write to a file. Listing 14-1 shows a rudimentary program that reads integers from a file named *list1401.txt* and writes them, one per line, to the standard output stream. If the program cannot open the file, it prints an error message.

Listing 14-1. Copying Integers from a File to Standard Output

```
#include <cstdio>
#include <fstream>
#include <iostream>

int main()
{
  std::ifstream in{"list1401.txt"};
  if (not in)
    std::perror("list1401.txt");
  else
  {
    int x{};
    while (in >> x)
      std::cout << x << '\n';
    in.close();

  }
}
```

The `<fstream>` header declares `ifstream`, which is the type you use to read from a file. To open a file, simply name the file in `ifstream`'s initializer. If the file cannot be opened, the `ifstream` object is left in an error state, a condition for which you can test using an `if` statement. When you are done reading from the file, call the `close()` member function. After closing the stream, you cannot read from it anymore.

Once the file is open, read from it the same way you read from `std::cin`. All the input operators that are declared in `<istream>` work equally well for an `ifstream`, as they do for `std::cin`.

The `std::perror` (declared in `<cstdio>`) function prints an error message. If the file cannot be opened, the exact reason is saved in a global variable, and `perror` uses that variable to decide which message to print. It also prints its argument.

Run the program when you know the input file does not exist. What message does the program display?

____

If you can, create the input file, then change the protection on the file, so you can no longer read it. Run the program.

What message do you get this time?

____

Writing Files

As you have probably guessed, to write to a file, you define an `ofstream` object. To open the file, simply name the file in the variable's initializer. If the file does not exist, it will be created. If the file does exist, its old contents are discarded in preparation for writing new contents. If the file cannot be opened, the `ofstream` object is left in an error state, so remember to test it before you try to use it. Use an `ofstream` object the same way you use `std::cout`.

Modify Listing 14-1 to write the numbers to a named file. This time, name the input file *list1402.in* and name the output file *list1402.out*. Compare your solution with mine in Listing 14-2.

Listing 14-2. Copying Integers from a Named File to a Named File

```
#include <cstdio>
#include <fstream>
#include <iostream>

int main()
{
  std::ifstream in{"list1402.in"};
  if (not in)
    std::perror("list1402.in");
  else
  {
    std::ofstream out{"list1402.out"};
    if (not out)
      std::perror("list1402.out");
    else
    {
      int x{};
      while (in >> x)
        out << x << '\n';
      out.close();
      in.close();
    }
  }
}
```

Like `ifstream`, the `ofstream` type is declared in `<fstream>`.

The program opens the input file first. If that succeeds, it opens the output file. If the order were reversed, the program might create the output file then fail to open the input file, and the result would be a wasted, empty file. Always open input files first.

Also notice that the program does not close the input file, if it cannot open the output file. Don't worry: it closes the input file just fine. When `in` is destroyed at the end of `main`, the file is automatically closed.

I know what you're thinking: If `in` is automatically closed, why call `close` at all? Why not let `in` close automatically in all cases? For an input file, that's actually okay. Feel free to delete the `in.close();` statement from the program. For an output file, however, doing so is unwise.

Some output errors do not arise until the file is closed, and the operating system flushes all its internal buffers and does all the other cleanup it needs to do when closing a file. Thus, an output stream object might not receive an error from the operating system until you call `close()`. Detecting and handling these errors is an advanced skill. The first step toward developing that skill is to adopt the habit of calling `close()` explicitly for output files. When it comes time to add the error-checking, you will have a place where you can add it.

Try running the program in Listing 14-2 in various error scenarios. Create the output file, *list1402.out*, and then use the operating system to mark the file as read-only. **What happens?**

____

If you noticed that the program does not check whether the output operations succeed, congratulations for having sharp eyes! C++ offers a few different ways to check for output errors, but they all have drawbacks. The easiest is to test whether the output stream is in an error state. You can check the stream after every output operation, but that approach is cumbersome, and few people write code that way. Another way lets the stream check for an error condition after every operation and alerts your program with an exception. You'll learn about this technique in Exploration 45. A frighteningly common technique is to ignore output errors altogether. As a compromise, I recommend testing for errors after calling `close()`. Listing 14-3 shows the final version of the program.

Listing 14-3. Copying Integers, with Minimal Error-Checking

```
#include <cstdio>
#include <fstream>
#include <iostream>

int main()
{
  std::ifstream in{"list1402.in"};
  if (not in)
    std::perror("list1402.in");
  else
  {
    std::ofstream out{"list1402.out"};
    if (not out)
      std::perror("list1402.out");
    else
    {
      int x{};
      while (in >> x)
        out << x << '\n';
      out.close();
      if (not out)
        std::perror("list1402.out");
    }
  }
}
```

Basic I/O is not difficult, but it can quickly become a morass of gooey, complicated code when you start to throw in sophisticated error-handling, international issues, binary I/O, and so on. Later Explorations will introduce most of these topics, but only when the time is ripe. For now, however, go back to earlier programs and practice modifying them to read and write named files instead of the standard input and output streams. For the sake of brevity (if for no other reason), the examples in the book will continue to use the standard I/O streams. If your IDE interferes with redirecting the standard I/O streams, or if you just prefer named files, you now know how to change the examples to meet your needs.

EXPLORATION 15

The Map Data Structure

Now that you understand the basics, it's time to move on to more exciting challenges. Let's write a real program—something nontrivial but still simple enough to master this early in the book. Your task is to write a program that reads words and counts the frequency of each unique word. For the sake of simplicity, a word is a string of non-space characters separated by white space. Be aware, however, that by this definition, words end up including punctuation characters, but we'll worry about fixing that problem later.

This is a complicated program, touching on everything you've learned about C++ so far. If you want to exercise your new understanding of file I/O, read from a named file. If you prefer the simplicity, read from the standard input. Before jumping in and trying to write a program, take a moment to think about the problem and the tools you need to solve it. **Write pseudo-code for the program.** Try to write C++ code where you can, and make up whatever else you need to tackle the problem. Keep it simple—and don't dwell on trying to get syntax details correct.

Using Maps

The title of this Exploration tells you what C++ feature will help provide an easy solution to this problem. What C++ calls a *map*, some languages and libraries call a *dictionary* or *association*. A map is simply a data structure that stores pairs of keys and values, indexed by the key. In other words, it maps a key to a value. Within a map, keys are unique. The map stores keys in ascending order. Thus, the heart of the program is a map that stores strings as keys and number of occurrences as the associated value for each key.

Naturally, your program needs the `<map>` header. The map datatype is called `std::map`. To define a map, you need to specify the key and value type within angle brackets (separated by a comma), as shown in the following example:

```
std::map<std::string, int> counts;
```

You can use almost any type as the key and value types, even another map. As with `vector`, if you do not initialize a `map`, it starts out empty.

The simplest way to use a map is to look up values using square brackets. For example, `counts["the"]` returns the value associated with the key, `"the"`. If the key is not present in the map, it is added with an initial value of zero. If the value type were `std::string`, the initial value would be an empty string.

Armed with this knowledge, you can write the first part of the program—collecting the word counts—as shown in Listing 15-1. (Feel free to modify the program to read from a named file, as you learned in Exploration 14.)

Listing 15-1. Counting Occurrences of Unique Words

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> counts{};
  std::string word{};
  while (std::cin >> word)
    ++counts[word];
  // TODO: Print the results.
}
```

In Listing 15-1, the ++ operator increments the count that the program stores in `counts`. In other words, when `counts[word]` retrieves the associated value, it does so in a way that lets you modify the value. You can use it as a target for an assignment or apply the increment or decrement operator.

For example, suppose you wanted to reset a count to zero.

```
counts["something"] = 0;
```

That was easy. Now all that's left to do is to print the results. Like vectors, maps also use iterators, but because an iterator refers to a key/value pair, they are a little more complicated to use than a vector's iterators.

Pairs

The best way to print the map is to use a range-based `for` loop to iterate over the map. Each map element is a single object that contains the key and the value. The key is called `first`, and the value is called `second`.

■ **Note** The two parts of the map element's value are not named key and value, because the std::pair type is a generic part of the C++ library. The library uses this type in several different places. Thus, the names of the parts of a pair are also generic and not tied specifically to map.

Use a dot (.) operator to access a member of the pair. To keep things simple, print the output as the key, followed by a tab character, followed by the count, all on one line. Putting all these pieces together, you end up with the finished program, as presented in Listing 15-2.

Listing 15-2. Printing Word Frequencies

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
    ++counts[word];

  // For each word/count pair...
  for (auto element : counts)
    // Print the word, tab, the count, newline.
    std::cout << element.first << '\t' << element.second << '\n';
}
```

I snuck in another new feature: auto. The true type of element is

```
std::pair<const std::string, int>
```

But is the true type all that important? When iterating over the map, you know you will use the .first and .second members. Let the compiler worry about the details. That's what the auto keyword is for. It tells the compiler to figure out the type of element based on the iterator type of counts.

Using the knowledge you gained in Exploration 8, you know how to format the output as two neat columns. All that is required is to find the size of the longest key. In order to right-align the counts, you can try to determine the number of places required by the largest count, or you can simply use a very large number, such as 10.

Rewrite Listing 15-2 to line up the output neatly, according to the size of the longest key.

Naturally, you will need to write another loop to visit all the elements of counts and test the size of each element. In Exploration 10, you learned that vector has a size() member function that returns the number of elements in the vector. Would you be surprised to learn that map and string also have size() member functions? The designers of the C++ library did their best to be consistent with names. The size() member function returns an integer of type size_type.

■ **Tip** Remember `size_type` from Exploration 10? If not, go back and refresh your memory. Exploration 10 has some important admonitions about `size_type`.

Compare your program with Listing 15-3.

Listing 15-3. Aligning Words and Counts Neatly

```
#include <iomanip>
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
    ++counts[word];

  // Determine the longest word.
  std::string::size_type longest{};
  for (auto element : counts)
    if (element.first.size() > longest)
      longest = element.first.size();

  // For each word/count pair...
  const int count_size{10}; // Number of places for printing the count
  for (auto element : counts)
    // Print the word, count, newline. Keep the columns neatly aligned.
    std::cout << std::setw(longest)    << std::left  << element.first <<
            std::setw(count_size) << std::right << element.second << '\n';
}
```

If you want some sample input, try the file *explore15.txt*, which you can download from this book's web site. Notice how the word is left-aligned and the count is right-aligned. We expect numbers to be right-aligned, and words are customarily left-aligned (in Western cultures). And remember `const` from Exploration 8? That simply means `count_size` is a constant.

Searching in Maps

A `map` stores its data in sorted order by key. Searching in a `map`, therefore, is pretty fast (logarithmic time). Because a `map` keeps its keys in order, you can use any of the standard binary search algorithms (such as `lower_bound`, to which you were introduced in Exploration 13), but even better is to use `map`'s member functions. These member functions have the same names as the standard algorithms but can take advantage of their knowledge of a `map`'s internal structure. The member functions also run in logarithmic time, but with less overhead than the standard algorithms.

For example, suppose you want to know how many times the word *the* appears in an input stream. You can read the input and collect the counts in the usual way, then call `find("the")` to see if `"the"` is in the `map`, and if so, get an iterator that points to its key/value pair. If the key is not in the map, `find()` returns the `end()` iterator. If the key is present, you can extract the count. You have all the knowledge and skills you need to solve this problem, so go ahead and **write the program to print the number of occurrences of the word *the*.** Once again, you can use *explore15.txt* as sample input. If you don't want to use redirection, modify the program to read from the *explore15.txt* file.

What count does your program print when you provide this file as the input? __________ The program presented in Listing 15-4 detects ten occurrences.

Listing 15-4. Searching for a Word in a Map

```
#include <iomanip>
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
    ++counts[word];

  std::map<std::string,int>::iterator the{counts.find("the")};
  if (the == counts.end())
    std::cout << "\"the\": not found\n";
  else if (the->second == 1)
    std::cout << "\"the\": occurs " << the->second << " time\n";
  else
    std::cout << "\"the\": occurs " << the->second << " times\n";
}
```

Until now, you've used a dot (`.`) to access a member, such as `find()` or `end()`. An iterator is different. You have to use an arrow (`->`) to access a member from an iterator, hence `the->second`. You won't see this style much until Exploration 32.

I don't know about you, but I find `map<string, int>::iterator` to be unwieldy. That's what the `auto` keyword is for. So why didn't I use `auto` to declare the iterator, `the`? Here we run into a small problem. The `auto` keyword and universal initialization don't always play together the way you expect them to. **Try changing the iterator type to use auto. What happens?**

____

____

Ouch. Each compiler is different, but mine spews out a screenful of cryptic errors. In this one case, universal initialization fails us. Instead, you must use an equal sign or parentheses to initialize the `auto` iterator. Either of the following two variable definitions will work:

```
auto iterator_using_parens( counts.find("the") );
auto iterator_using_equal = counts.find("the");
```

If you don't like the exception in this case, C++ (like C) offers another way out: type synonyms, which just happens to be the subject of the next Exploration.

EXPLORATION 16

Type Synonyms

Using types such as `std::vector<std::string>::size_type` or `std::map<std::string, int>::iterator` can be clumsy, prone to typographical errors, and just plain annoying to type and read. Fortunately, C++ lets you define short synonyms for clumsy types. You can also use type synonyms to provide meaningful names for generic types. (The standard library has quite a few synonyms of the latter variety.) These synonyms are often referred to as typedefs, because the keyword you use to declare them is `typedef`.

typedef Declarations

C++ inherits the basic syntax and semantics of `typedef` from C, so you might already be familiar with this keyword. If so, please bear with me while I bring other readers up to speed.

The idea of a `typedef` is to create a synonym, or alias, for another type. There are two compelling reasons for creating type synonyms.

- They create a short synonym for a long type name. For example, you may want to use `count_iter` as a type synonym for `std::map<std::string,int>::iterator`.
- They create a mnemonic synonym. For example, a program might declare `height` as a synonym for `int`, to emphasize that variables of type `height` store a height value. This information helps the human reader understand the program.

The basic syntax for a `typedef` declaration is like defining a variable, except you start with the `typedef` keyword, and the name of the type synonym takes the place of the variable name.

```
typedef std::map<std::string,int>::iterator count_iter;
typedef int height;
```

Revisit Listing 15-4 and simplify the program by using a `typedef` declaration. Compare your result with Listing 16-1.

Listing 16-1. Counting Words, with a Clean Program That Uses `typedef`

```
#include <iomanip>
#include <iostream>
#include <map>
#include <string>
```

```
int main()
{
  std::map<std::string, int> counts{};
  typedef std::map<std::string,int> count_map;
  typedef count_map::iterator count_iterator;

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
    ++counts[word];

  count_iterator the{counts.find("the")};

  if (the == counts.end())
    std::cout << "\"the\": not found\n";
  else if (the->second == 1)
    std::cout << "\"the\": occurs " << the->second << " time\n";
  else
    std::cout << "\"the\": occurs " << the->second << " times\n";
}
```

I like the new version of this program. It's a small difference in this little program, but it lets us continue to use universal initialization, while preserving clarity and readability.

Common typedefs

The standard library makes heavy use of typedefs, as you have already seen. For example, `std::vector<int>::size_type` is a typedef for an integer type. You don't know which integer type (C++ has several, which you will learn about in Exploration 25), nor does it matter. All you have to know is that `size_type` is the type to use if you want to store a size or index in a variable.

Most likely, `size_type` is a typedef for `std::size_t`, which is itself a typedef. The `std::size_t` typedef is a synonym for an integer type that is suitable for representing a size. In particular, C++ has an operator, `sizeof`, which returns the size in bytes of a type or object. The result of `sizeof` is an integer of type `std::size_t`, however a compiler-writer chooses to implement `sizeof` and `std::size_t`.

■ **Note** A "byte" is defined as the size of type `char`. So, by definition, `sizeof(char)` == 1. The size of other types depends on the implementation. On most popular desktop workstations, `sizeof(int)` == 4, but 2 and 8 are also likely candidates.

Type Aliases

Another way to declare a typedef is with the using keyword. This style of type declaration is called a type alias. For example:

```
using height = int;
using count_map = std::map<std::string, int>;
```

The meaning of a type alias is identical to a typedef. They are merely two different syntaxes for the same concept. Some people find type aliases to be easier to read, because the name you are declaring comes first, rather than being hidden in a typedef declaration.

That was short and sweet, wasn't it? Now you can return to the problem of counting words. This program has a number of usability flaws.

What can you think of to improve the word-counting program?

____

____

____

At the top of my list are the following two items:

- Ignore punctuation marks.
- Ignore case differences.

In order to implement these additional features, you have to learn some more C++. For example, the C++ standard library has functions to test whether a character is punctuation, a digit, an uppercase letter, a lowercase letter, and so on. The next Exploration begins by exploring characters more closely.

EXPLORATION 17

Characters

In Exploration 2, I introduced you to character literals in single quotes, such as `'\n'`, to end a line of output, but I have not yet taken the time to explain these fundamental building blocks. Now is the time to explore characters in greater depth.

Character Type

The `char` type represents a single character. Internally, all computers represent characters as integers. The *character set* defines the mapping between characters and numeric values. Common character sets are ISO 8859-1 (also called Latin-1) and ISO 10646 (same as Unicode), but many, many other character sets are in wide use.

The C++ standard does not mandate any particular character set. The literal `'4'` represents the digit 4, but the actual value that the computer uses internally is up to the implementation. You should not assume any particular character set. For example, in ISO 8859-1 (Latin-1), `'4'` has the value 52, but in EBCDIC, it has the value 244.

Similarly, given a numeric value, you cannot assume anything about the character that value represents. If you know a `char` variable stores the value 169, the character may be `'z'` (EBCDIC), `'©'` (Unicode), or `'Љ'` (ISO 8859-5).

C++ does not try to hide the fact that a character is actually a number. You can compare `char` values with `int` values, assign a `char` to an `int` variable, or do arithmetic with `char`s. Performing arithmetic is fraught with danger, unless you know the actual character set, but C++ guarantees that any character set your compiler and library support represents digit characters with contiguous values, starting at `'0'`. Thus, for example, the following is true for all C++ implementations:

```
'0' + 7 == '7'
```

Read Listing 17-1. **What does the program do?** (Hint: the `get` member function reads a single character from the stream. It does not skip over white space or treat any character specially. Extra hint: what happens if you subtract `'0'` from a character that you know to be a digit?)

____

____

____

____

Listing 17-1. Working and Playing with Characters

```
#include <iostream>

int main()
{
  int value{};
  bool have_value{false};
  char ch{};

  while (std::cin.get(ch))
  {
    if (ch >= '0' and ch <= '9')
    {
      value = ch - '0';
      have_value = true;
      while (std::cin.get(ch) and ch >= '0' and ch <= '9')
        value = value * 10 + ch - '0';

    }
    if (ch == '\n')
    {
      if (have_value)
      {
        std::cout << value << '\n';
        have_value = false;
      }
    }
    else if (ch != ' ' and ch != '\t')
    {
      std::cout << '\a';
      have_value = false;

      while (std::cin.get(ch) and ch != '\n')
        /*empty*/;
    }
  }
}
```

Briefly, this program reads numbers from the standard input and echoes the values to the standard output. If the program reads any invalid characters, it alerts the user (with `\a`, which I describe later in this Exploration), ignores the line of input, and discards the value. Leading and trailing blank and tab characters are allowed. The program prints the saved numeric value only after reaching the end of an input line. This means if a line contains more than one valid number, the program prints only the last value. I ignore the possibility of overflow, to keep the code simple.

The `get` function takes a character variable as an argument. It reads one character from the input stream, then stores the character in that variable. The `get` function does not skip over white space. When you use `get` as a loop condition, it returns `true`, if it successfully reads a character, and the program should keep reading. It returns `false` if no more input is available or some kind of input error occurred.

All the digit characters have contiguous values, so the inner loop tests to determine if a character is a digit character by comparing it to the values for `'0'` and `'9'`. If it is a digit, subtracting the value of `'0'` from it leaves you with an integer in the range 0 to 9.

The final loop reads characters and does nothing with them. The loop terminates when it reads a new line character. In other words, the final loop reads and ignores the rest of the input line.

Programs that need to handle white space on their own (such as Listing 17-1) can use `get`, or you can tell the input stream not to skip over white space prior to reading a number or anything else. The next section discusses character I/O in more detail.

Character I/O

You just learned that the `get` function reads a single character without treating white space specially. You can do the same thing with a normal input operator, but you must use the `std::noskipws` manipulator. To restore the default behavior, use the `std::skipws` manipulator (declared in `<ios>`).

```
// Skip white space, then read two adjacent characters.
char left, right;
std::cin >> left >> std::noskipws >> right >> std::skipws;
```

After turning off the `skipws` flag, the input stream does not skip over leading whitespace characters. For instance, if you were to try to read an integer, and the stream is positioned at white space, the read would fail. If you were to try to read a string, the string would be empty, and the stream position would not advance. So you have to consider carefully whether to skip white space. Typically, you would do that only when reading individual characters.

Remember that an input stream uses the `>>` operator (Exploration 5), even for manipulators. Using `>>` for manipulators seems to break the mnemonic of transferring data to the right, but it follows the convention of always using `>>` with an input stream. If you forget, the compiler will remind you.

Write a program that reads the input stream one character at a time and echoes the input to the standard output stream verbatim. This is not a demonstration of how to copy streams but an example of working with characters. Compare your program with Listing 17-2.

Listing 17-2. Echoing Input to Output, One Character at a Time

```
#include <iostream>

int main()
{
  std::cin >> std::noskipws;
  char ch;
  while (std::cin >> ch)
    std::cout << ch;
}
```

You can also use the `get` member function, in which case you don't need the `noskipws` manipulator.

Let's try something a little more challenging. Suppose you have to read a series of points. The points are defined by a pair of *x*, *y* coordinates, separated by a comma. White space is allowed before and after each number and around the comma. Read the points into a vector of *x* values and a vector of *y* values. Terminate the input loop if a point does not have a proper comma separator. Print the vector contents, one point per line. I know this is a bit dull, but the point is to experiment with character input. If you prefer, do something special with the data. Compare your result with Listing 17-3.

Listing 17-3. Finding the Points with the Largest x and y Values

```
#include <algorithm>
#include <iostream>
#include <limits>
#include <vector>

int main()
{
  typedef std::vector<int>  intvec;
  typedef intvec::iterator  iterator;

  intvec xs, ys;        // store the xs and ys

  { // local block to keep I/O variables local
    int x{}, y{};      // variables for I/O
    char sep{};
    // Loop while the input stream has an integer (x), a character (sep),
    // and another integer (y); then test that the separator is a comma.
    while (std::cin >> x >> sep and sep == ',' and std::cin >> y)
    {
      xs.push_back(x);
      ys.push_back(y);
    }
  }

  for (iterator x{xs.begin()}, y{ys.begin()}; x != xs.end(); ++x, ++y)
    std::cout << *x << ',' << *y << '\n';
}
```

The `while` loop is the key. The loop condition reads an integer and a character and tests to determine if the character is a comma, before reading a second integer. The loop terminates if the input is invalid or ill-formed or if the loop reaches the end of file. A more sophisticated program would distinguish between these two cases, but that's a side issue for the moment.

Also notice how the final `for` loop iterates over two vectors at the same time. A range-based `for` loop doesn't help in this case, so the loop must use explicit iterators. Notice how a typedef keeps the `for` loop header tidy and readable. (Recall from Exploration 15 that the `auto` keyword doesn't work as expected when combined with universal initialization.)

Newlines and Portability

You've probably noticed that Listing 17-3, and every other program I've presented so far, prints `'\n'` at the end of each line of output. We have done so without considering what this really means. Different environments have different conventions for end-of-line characters. UNIX uses a line feed (`'\x0a'`); MacOS uses a carriage return (`'\x0d'`); DOS and Microsoft Windows use a combination of a carriage return followed by a line feed (`'\x0d\x0a'`); and some operating systems don't use line terminators but, instead, have record-oriented files, in which each line is a separate record.

In all these cases, the C++ I/O streams automatically convert a native line ending to a single `'\n'` character. When you print `'\n'` to an output stream, the library automatically converts it to a native line ending (or terminates the record).

In other words, you can write programs that use `'\n'` as a line ending and not concern yourself with native OS conventions. Your source code will be portable to all C++ environments.

Character Escapes

In addition to `'\n'`, C++ offers several other *escape sequences*, such as `'\t'`, for horizontal tab. Table 17-1 lists all the character escapes. Remember that you can use these escapes in character literals and string literals.

Table 17-1. *Character Escape Sequences*

Escape	Meaning
`\a`	Alert: ring a bell or otherwise signal the user
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Literal `\`
`\?`	Literal `?`
`\'`	Literal `'`
`\"`	Literal `"`
`\x`*XX*`...`	Hexadecimal (base 16) character value

The last two items are the most interesting. An escape sequence of one to three octal digits (`0` to `7`) specifies the value of the character. Which character the value represents is up to the implementation.

Understanding all the caveats from the first section of this Exploration, there are times when you must specify an actual character value. The most common is `'\0'`, which is the character with value zero, also called a *null character*, which you may utilize to initialize `char` variables. It has some other uses as well, especially when interfacing with C functions and the C standard library.

The final escape sequence (`\x`) lets you specify a character value in hexadecimal. Typically, you would use two hexadecimal digits, because this is all that fits in the typical, 8-bit `char`. (The purpose of longer `\x` escapes is for wide characters, the subject of Exploration 55.)

The next Exploration continues your understanding of characters by examining how C++ classifies characters according to letter, digit, punctuation, etc.

EXPLORATION 18

Character Categories

Exploration 17 introduced and discussed characters. This Exploration continues that discussion with character classification (e.g., upper or lowercase, digit or letter), which, as you will see, turns out to be more complicated than you might have expected.

Character Sets

As you learned in Exploration 17, the numeric value of a character, such as `'A'`, depends on the character set. The compiler must decide which character set to use at compile time and at runtime. This is typically based on preferences that the end user selects in the host operating system.

Character-set issues rarely arise for the basic subset of characters—such as letters, digits, and punctuation symbols—that are used to write C++ source code. Although it is conceivable that you could compile a program using ISO 8859-1 and run that program using EBCDIC, you would have to work pretty hard to arrange such a feat. Most likely, you will find yourself using one or more character sets that share some common characteristics. For example, all ISO 8859 character sets use the same numeric values for the letters of the Roman alphabet, digits, and basic punctuation. Even most Asian character sets preserve the values of these basic characters.

Thus, most programmers blithely ignore the character-set issue. We use character literals, such as `'%'` and assume the program will function the way we expect it to, on any system, anywhere in the world—and we are usually right. But not always.

Assuming the basic characters are always available in a portable manner, we can modify the word-counting program to treat only letters as characters that make up a word. The program would no longer count `right` and `right?` as two distinct words. The `string` type offers several member functions that can help us search in strings, extract substrings, and so on.

For example, you can build a string that contains only the letters and any other characters that you want to consider to be part of a word (such as `'-'`). After reading each word from the input stream, make a copy of the word but keep only the characters that are in the string of acceptable characters. Use the `find` member function to try to find each character; `find` returns the zero-based index of the character, if found, or `std::string::npos`, if not found.

Using the find function, rewrite Listing 15-3 to clean up the word string prior to inserting it in the map. Test the program with a variety of input samples. How well does it work? Compare your program with Listing 18-1.

Listing 18-1. Counting Words: Restricting Words to Letters and Letter-Like Characters

```
#include <iomanip>
#include <iostream>
#include <map>
#include <string>
```

```
int main()
{
  typedef std::map<std::string, int>    count_map;
  typedef std::string::size_type        str_size;

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string okay{"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                   "abcdefghijklmnopqrstuvwxyz"
                   "0123456789-_"};
  std::string word{};
  while (std::cin >> word)
  {
    // Make a copy of word, keeping only the characters that appear in okay.
    std::string copy{};
    for (char ch : word)
      if (okay.find(ch) != std::string::npos)
        copy.push_back(ch);
    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }

  // Determine the longest word.
  str_size longest{0};
  for (auto pair : counts)
    if (pair.first.size() > longest)
      longest = pair.first.size();

  // For each word/count pair...
  const int count_size{10}; // Number of places for printing the count
  for (auto pair : counts)
    // Print the word, count, newline. Keep the columns neatly aligned.
    std::cout << std::setw(longest)    << std::left << pair.first <<
                 std::setw(count_size) << std::right << pair.second << '\n';
}
```

Some of you may have written a program very similar to mine. Others among you—particularly those living outside the United States—may have written a slightly different program. Perhaps you included other characters in your string of acceptable characters.

For example, if you are French and using Microsoft Windows (and the Windows 1252 character set), you may have defined the okay object as follows:

```
std::string okay{"ABCDEFGHIJKLMNOPQRSTUVWXYZÀÁÄÇÈÉÊËÎÏÔÙÛÜŒŸ"
            "abcdefghijklmnopqrstuvwxyzàáäçèéêëîïöùûüœÿ"
            "0123456789-_"};
```

But what if you then try to compile and run this program in a different environment, particularly one that uses the ISO 8859-1 character set (popular on UNIX systems)? ISO 8859-1 and Windows 1252 share many character codes but differ in a few significant ways. In particular, the characters `'Œ'`, `'œ'`, and `'Ÿ'` are missing from ISO 8859-1. As a result, the program may not compile successfully in an environment that uses ISO 8859-1 for the compile-time character set.

What if you want to share the program with a German user? Surely that user would want to include characters such as `'Ö'`, `'ö'`, and `'ß'` as letters. What about Greek, Russian, and Japanese users?

We need a better solution. Wouldn't it be nice if C++ provided a simple function that would notify us if a character is a letter, without forcing us to hard-code exactly which characters are letters? Fortunately, it does.

Character Categories

An easier way to write the program in Listing 18-1 is to call the `isalnum` function (declared in `<locale>`). This function indicates whether a character is alphanumeric in the runtime character set. The advantage of using `isalnum` is that you don't have to enumerate all the possible alphanumeric characters; you don't have to worry about differing character sets; and you don't have to worry about accidentally omitting a character from the approved string.

Rewrite Listing 18-1 to call isalnum instead of find. The first argument to `std::isalnum` is the character to test, and the second is `std::locale{""}`. (Don't worry yet about what that means. Have patience: I'll get to that soon.)

Try running the program with a variety of alphabetic input, including accented characters. Compare the results with the results from your original program. The files that accompany this book include some samples that use a variety of character sets. Choose the sample that matches your everyday character set and run the program again, redirecting the input to that file.

If you need help with the program, see my version of the program in Listing 18-2. For the sake of brevity, I eliminated the neat-output part of the code, reverting to simple strings and tabs. Feel free to restore the pretty output, if you desire.

Listing 18-2. Testing a Character by Calling `std::isalnum`

```
#include <iostream>
#include <locale>
#include <map>
#include <string>

int main()
{
  typedef std::map<std::string, int>    count_map;

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
  {
    // Make a copy of word, keeping only alphabetic characters.
    std::string copy{};
    for (char ch : word)
      if (std::isalnum(ch, std::locale{""}))
        copy.push_back(ch);
    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }
  // For each word/count pair, print the word & count on one line.
  for (auto pair : counts)
    std::cout << pair.first << '\t' << pair.second << '\n';
}
```

Now turn your attention to the `std::locale{""}` argument. The locale directs `std::isalnum` to the character set it should use to test the character. As you saw in Exploration 17, the character set determines the identity of a character, based on its numeric value. A user can change character sets while a program is running, so the program must keep track of the user's actual character set and not depend on the character set that was active when you compiled the program.

Download the files that accompany this book and find the text files whose names begin with `sample`. **Find the one that best matches the character set you use every day, and select that file as the redirected input to the program.** Look for the appearance of the special characters in the output.

Change `locale{""}` to `locale{}` in the boldface line of Listing 18-2. Now compile and run the program with the same input. **Do you see a difference?** ________________ **If so, what is the difference?**

____

____

Without knowing more about your environment, I can't tell you what you should expect. If you are using a Unicode character set, you won't see any difference. The program would not treat any of the special characters as letters, even when you can plainly see they are letters. This is due to the way Unicode is implemented, and Exploration 55 will discuss this topic in depth.

Other users will notice that only one or two strings make it to the output. Western Europeans who use ISO 8859-1 may notice that ÁÇÐÈ is considered a word. Greek users of ISO 8859-7 will see ΑΒΓΔΕ as a word.

Power users who know how to change their character sets on the fly can try several different options. You must change the character set that programs use at runtime and the character set that your console uses to display text.

What is most noticeable is that the characters the program considers to be letters vary from one character set to another. But after all, that's the idea of different character sets. The knowledge of which characters are letters in which character sets is embodied in the locale.

Locales

In C++, a *locale* is a collection of information pertaining to a culture, region, and language. The locale includes information about

- formatting numbers, currency, dates, and time
- classifying characters (letter, digit, punctuation, etc.)
- converting characters from uppercase to lowercase and *vice versa*
- sorting text (e.g., is `'A'` less than, equal to, or greater than `'Å'`?)
- message catalogs (for translations of strings that your program uses)

Every C++ program begins with a minimal, standard locale, which is known as the *classic* or `"C"` locale. The `std::locale::classic()` function returns the classic locale. The unnamed locale, (`std::locale{""}`), is the user's preferences that C++ obtains from the host operating system. The locale with the empty-string argument is often known as the *native* locale.

The advantage of the classic locale is that its behavior is known and fixed. If your program must read data in a fixed format, you don't want the user's preferences getting in the way. By contrast, the advantage of the native format is that the user chose those preferences for a reason and wants to see program output follow that format. A user who always specifies a date as day/month/year doesn't want a program printing month/day/year simply because that's the convention in the programmer's home country.

Thus, the classic format is often used for reading and writing data files, and the native format is best used to interpret input from the user and to present output directly to the user.

Every I/O stream has its own `locale` object. To affect the stream's `locale`, call its `imbue` function, passing the `locale` object as the sole argument.

■ **Note** You read that correctly: `imbue`, not `setlocale` or `setloc`—given that the `getloc` function returns a stream's current locale—or anything else that might be easy to remember. On the other hand, `imbue` is such an unusual name for a member function, you may remember it for that reason alone.

In other words, when C++ starts up, it initializes each stream with the classic locale, as follows:

```
std::cin.imbue(std::locale::classic());
std::cout.imbue(std::locale::classic());
```

Suppose you want to change the output stream to adopt the user's native locale. Do this using the following statement at the start of your program:

```
std::cout.imbue(std::locale{""});
```

For example, suppose you have to write a program that reads a list of numbers from the standard input and computes the sum. The numbers are raw data from a scientific instrument, so they are written as digit strings. Therefore, you should continue to use the classic locale to read the input stream. The output is for the user's benefit, so the output should use the native locale.

Write the program and try it with very large numbers, so the output will be greater than 1000. What does the program print as its output? ________________

See Listing 18-3 for my approach to solving this problem.

Listing 18-3. Using the Native Locale for Output

```
#include <iostream>
#include <locale>

int main()
{
  std::cout.imbue(std::locale{""});

  int sum{0};
  int x{};
  while (std::cin >> x)
    sum = sum + x;
  std::cout << "sum = " << sum << '\n';
}
```

When I run the program in Listing 18-3 in my default locale (United States), I get the following result:

```
sum = 1,234,567
```

Notice the commas that separate thousands. In some European countries, you might see the following instead:

```
sum = 1.234.567
```

You should obtain a result that conforms to native customs, or at least follows the preferences that you set in your host operating system.

When you use the native locale, I recommend defining a variable of type `std::locale` in which to store it. You can pass this variable to `isalnum`, `imbue`, or other functions. By creating this variable and distributing copies of it, your program has to query the operating system for your preferences only once, not every time you need the `locale`. Thus, the main loop ends up looking something like Listing 18-4.

Listing18-4. Creating and Sharing a Single Locale Object

```
#include <iostream>
#include <locale>
#include <map>
#include <string>

int main()
{
  typedef std::map<std::string, int>    count_map;

  std::locale native{""};             // get the native locale
  std::cin.imbue(native);             // interpret the input and output according to
  std::cout.imbue(native);            // the native locale

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
  {
    // Make a copy of word, keeping only alphabetic characters.
    std::string copy{};
    for (char ch : word)
      if (std::isalnum(ch, native))
        copy.push_back(ch);
    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }

  // For each word/count pair, print the word & count on one line.
  for (auto pair : counts)
    std::cout << pair.first << '\t' << pair.second << '\n';
}
```

The next step toward improving the word-counting program is to ignore case differences, so the program does not count the word The as different from `the`. It turns out this problem is trickier than it first appears, so it deserves an entire Exploration of its own.

EXPLORATION 19

Case-Folding

Picking up where we left off in Exploration 18, the next step to improving the word-counting program is to update it, so that it ignores case differences when counting. For example, the program should count `The` just as it does `the`. This is a classic problem in computer programming. C++ offers some rudimentary help but lacks some important fundamental pieces. This Exploration takes a closer look at this deceptively tricky issue.

Simple Cases

Western European languages have long made use of capital (or majuscule) letters and minuscule letters. The more familiar terms—uppercase and lowercase—arise from the early days of typesetting, when the type slugs for majuscule letters were kept in the upper cases of large racks containing all the characters used to make a printing plate. Beneath them were the cases, or boxes, that stored the minuscule letter slugs.

In the `<locale>` header, C++ declares the `isupper` and `islower` functions. They take a character as the first argument and a `locale` as the second argument. The return value is a `bool`: `true` if the character is uppercase (or lowercase, respectively) and `false` if the character is lowercase (or uppercase) or not a letter.

```
std::isupper('A', std::locale{""}) == true
std::islower('A', std::locale{""}) == false
std::isupper('Æ', std::locale{""}) == true
std::islower('Æ', std::locale{""}) == false
std::islower('½', std::locale{""}) == false
std::isupper('½', std::locale{""}) == false
```

The `<locale>` header also declares two functions to convert case: `toupper` converts lowercase to uppercase. If its character argument is not a lowercase letter, `toupper` returns the character as is. Similarly, `tolower` converts to lowercase, if the character in question is an uppercase letter. Just like the category testing functions, the second argument is a `locale` object.

Now you can **modify the word-counting program to fold uppercase to lowercase and count all words in lowercase.** Modify your program from Exploration 18, or start with Listing 18-4. If you have difficulty, take a look at Listing 19-1.

Listing 19-1. Folding Uppercase to Lowercase Prior to Counting Words

```
#include <iostream>
#include <locale>
#include <map>
#include <string>
```

```
int main()
{
  typedef std::map<std::string, int>    count_map;

  std::locale native{""};             // get the native locale
  std::cin.imbue(native);             // interpret the input and output according to
  std::cout.imbue(native);            // the native locale

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
  {
    // Make a copy of word, keeping only alphabetic characters.
    std::string copy{};
    for (char ch : word)
      if (std::isalnum(ch, native))
        copy.push_back(tolower(ch, native));
    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }

  // For each word/count pair, print the word & count on one line.
  for (auto pair : counts)
    std::cout << pair.first << '\t' << pair.second << '\n';
}
```

That was easy. So what's the problem?

Harder Cases

Some of you—especially German readers—already know the problem. Several languages have letter combinations that do not map easily between upper and lowercase, or one character maps to two characters. The German *Eszett*, ß, is a lowercase letter; when you convert it to uppercase, you get two characters: SS. Thus, if your input file contains "ESSEN" and "eßen", you want them to map to the same word, so they're counted together, but that just isn't feasible with C++. The way the program currently works, it maps "ESSEN" to "essen", which it counts as a different word from "eßen". A naïve solution would be to map "essen" to "eßen", but not all uses of ss are equivalent to ß.

Greek readers are familiar with another kind of problem. Greek has two forms of lowercase sigma: use ς at the end of a word and σ elsewhere. Our simple program maps Σ (uppercase sigma) to σ, so some words in all uppercase will not convert to a form that matches its lowercase version.

Sometimes, accents are lost during conversion. Mapping é to uppercase usually yields É but may also yield E. Mapping uppercase to lowercase has fewer problems, in that É maps to é, but what if that E (which maps to e) really means É, and you want it to map to é? The program has no way of knowing the writer's intentions, so all it can do is map the letters it receives.

Some character sets are more problematic than others. For example, ISO 8859-1 has a lowercase ÿ but not an uppercase equivalent (Ÿ). Windows 1252, on the other hand, extends ISO 8859-1, and one of the new code points is Ÿ.

■ **Tip** *Code point* is a fancy way of saying "numeric value that represents a character." Although most programmers don't use *code point* in everyday speech, those programmers who work closely with character set issues use it all the time, so you may as well get used to it. Mainstream programmers should become more accustomed to using this phrase.

In other words, converting case is impossible to do correctly using only the standard C++ library.

If you know your alphabet is one that C++ handles correctly, then go ahead and use `toupper` and `tolower`. For example, if you are writing a command line interpreter, within which you have full control over the commands, and you decide that the user should be able to enter commands in any case, simply make sure the commands map correctly from one case to another. This is easy to do, as all character sets can map the 26 letters of the Roman alphabet without any problems.

On the other hand, if your program accepts input from the user and you want to map that input to uppercase or lowercase, you cannot and must not use standard C++. For example, if you are writing a word processor, and you decide you need to implement some case-folding functions, you must write or acquire a non-standard function to implement the case-folding logic correctly. Most likely, you would need a library of character and string functions to implement your word processor. Case-folding would simply be one small part of this hypothetical library. (See this book's web site for some links to non-hypothetical libraries that can help you.)

What about our simple program? It isn't always practical to handle the full, complete, correct handling of cases and characters when you just want to count a few words. The case-handling code would dwarf the word-counting code.

In this case (pun intended), you must accept the fact that your program will sometimes produce incorrect results. Our poor little program will never recognize that "`ESSEN`" and "`eßen`" are the same word but in different cases. You can work around some of the multiple mappings (such as with Greek sigma) by mapping to uppercase then to lowercase. On the other hand, this can introduce problems with some accented characters. And I still have not touched upon the issue of whether "`naïve`" is the same word as "`naive`." In some locales, the diacritics are significant, which would cause "`naïve`" and "`naive`" to be interpreted as two different words. In other locales, they are the same word and should be counted together.

In some character sets, accented characters can be composed from separate non-accented characters followed by the desired accent. For example, you can write "`nai¨ve`", which is the same as "`naïve`."

I hope by now you are completely scared away from manipulating cases and characters. Far too many naïve programmers become entangled in this web, or worse, simply write bad code. I was tempted to wait until much later in the book before throwing all this at you, but I know that many readers will want to improve the word-counting program by ignoring case, so I decided to tackle the problem early.

Well, now you know better.

That doesn't mean you can't keep working on the word-counting program. The next Exploration returns to the realm of the realistic and feasible, as I finally show you how to write your own functions.

EXPLORATION 20

Writing Functions

At last, it's time to embark on the journey toward writing your own functions. In this Exploration, you'll begin by improving the word-counting program you've been crafting over the past five Explorations, writing functions to implement separate aspects of the program's functionality.

Functions

You've been using functions since the very first program you wrote. In fact, you've been writing functions too. You see, `main()` is a function, and you should view it the same as you would any other function (well, sort of, `main()` actually has some key differences from ordinary functions, but they needn't concern you yet).

A function has a return type, a name, and parameters in parentheses. Following that is a compound statement, which is the function body. If the function has no parameters, the parentheses are empty. Each parameter is like a variable declaration: type and name. Parameters are separated by commas, so you cannot declare two parameters after a single type name. Instead, you must specify the type explicitly for each parameter.

A function usually has at least one `return` statement, which causes the function to discontinue execution and return control to its caller. A `return` statement's structure begins with the `return` keyword, followed by an expression, and ends with a semicolon, as demonstrated in the following example:

```
return 42;
```

You can use a `return` statement anywhere you need a statement, and you can use as many `return` statements as you need or want. The only requirement is that every execution path through the function must have a `return` statement. Many compilers will warn you if you forget.

Some languages distinguish between functions, which return a value, and procedures or subroutines, which do not. C++ calls them all functions. If a function has no return value, declare the return type as `void`. Omit the value in the `return` statements in a `void` function:

```
return;
```

You can also omit the `return` statement entirely, if the function returns `void`. In this circumstance, control returns to the caller when execution reaches the end of the function body. Listing 20-1 presents some function examples.

Listing 20-1. Examples of Functions

```
#include <iostream>
#include <string>

/** Ignore the rest of the input line. */
void ignore_line()
```

```
{
  char c{};
  while (std::cin.get(c) and c != '\n')

    /*empty*/;
}

/** Prompt the user, then read a number, and ignore the rest of the line.
 * @param prompt the prompt string
 * @return the input number or 0 for end-of-file
 */
int prompted_read(std::string prompt)
{
  std::cout << prompt;
  int x{0};
  std::cin >> x;
  ignore_line();
  return x;
}

/** Print the statistics.
 * @param count the number of values
 * @param sum the sum of the values
 */
void print_result(int count, int sum)
{
  if (count == 0)
  {
    std::cout << "no data\n";
    return;
  }

  std::cout << "\ncount = " << count;
  std::cout << "\nsum   = " << sum;
  std::cout << "\nmean  = " << sum/count << '\n';
}

/** Main program.
 * Read integers from the standard input and print statistics about them.
 */
int main()
{
  int sum{0};
  int count{0};

  while (std::cin)
  {
    int x{ prompted_read("Value: ") };
    if (std::cin)
    {
      sum = sum + x;
      ++count;
    }
  }
  print_result(count, sum);
}
```

What does Listing 20-1 do?

____

____

The `ignore_line` function reads and discards characters from `std::cin` until it reaches the end of the line or the end of the file. It takes no arguments and returns no values to the caller.

The `prompted_read` function prints a prompt to `std::cout`, then reads a number from `std::cin`. It then discards the rest of the input line. Because `x` is initialized to `0`, if the read fails, the function returns `0`. The caller cannot distinguish between a failure and a real `0` in the input stream, so the `main()` function tests `std::cin` to know when to terminate the loop. (The value `0` is unimportant; feel free to initialize `x` to any value.) The sole argument to the function is the prompt string. The return type is `int`, and the return value is the number read from `std::cin`.

The `print_result` function takes two arguments, both of type `int`. It returns nothing; it simply prints the results. Notice how it returns early if the input contains no data.

Finally, the `main()` function puts it all together, repeatedly calling `prompted_read` and accumulating the data. Once the input ends, `main()` prints the results, which, in this example, are the sum, count, and average of the integers it read from the standard input.

Function Call

In a function call, all arguments are evaluated before the function is called. Each argument is copied to the corresponding parameter in the function, then the function body begins to run. When the function executes a `return` statement, the value in the statement is copied back to the caller, which can then use the value in an expression, assign it to a variable, and so on.

In this book, I try to be careful about terminology: *arguments* are the expressions in a function call, and *parameters* are the variables in a function's header. I've also seen the phrase *actual argument* used for arguments and *formal argument* used for parameters. I find these confusing, so I recommend you stick with the terms *arguments* and *parameters*.

Declarations and Definitions

I wrote the functions in bottom-up fashion because C++ has to know about a function before it can compile any call to that function. The easiest way to achieve this in a simple program is to write every function before you call it—that is, write the function earlier in the source file than the point at which you call the function.

If you prefer, you can code in a top-down manner and write `main()` first, followed by the functions it calls. The compiler still has to know about the functions before you call them, but you don't have to provide the complete function. Instead, you provide only what the compiler requires: the return type, name, and a comma-separated list of parameters in parentheses. Listing 20-2 shows this new arrangement of the source code.

Listing 20-2. Separating Function Declarations from Definitions

```
#include <iostream>
#include <string>

void ignore_line();
int prompted_read(std::string prompt);
void print_result(int count, int sum);

/** Main program.
 * Read integers from the standard input and print statistics about them.
 */
int main()
{
  int sum{0};
  int count{0};

  while (std::cin)
  {
    int x{ prompted_read("Value: ") };
    if (std::cin)
    {
      sum = sum + x;
      ++count;
    }
  }
  print_result(count, sum);
}

/** Prompt the user, then read a number, and ignore the rest of the line.
 * @param prompt the prompt string
 * @return the input number or -1 for end-of-file
 */
int prompted_read(std::string prompt)
{
  std::cout << prompt;
  int x{-1};
  std::cin >> x;
  ignore_line();
  return x;
}

/** Ignore the rest of the input line. */
void ignore_line()
{
  char c{};
  while (std::cin.get(c) and c != '\n')
    /*empty*/;
}
```

```
/** Print the statistics.
 * @param count the number of values
 * @param sum the sum of the values
 */
void print_result(int count, int sum)
{
  if (count == 0)
  {
    std::cout << "no data\n";
    return;
  }

  std::cout << "\ncount = " << count;
  std::cout << "\nsum   = " << sum;
  std::cout << "\nmean  = " << sum/count << '\n';
}
```

Writing the function in its entirety is known as providing a *definition*. Writing the function header by itself—that is, the return type, name, and parameters, followed by a semicolon—is known as a *declaration*. In general, a declaration tells the compiler how to use a name: what part of a program the name is (typedef, variable, function, etc.), the type of the name, and anything else (such as function parameters) that the compiler requires in order to make sure your program uses that name correctly. The definition provides the body or implementation for a name. A function's declaration must match its definition: the return types, name, and the types of the parameters must be the same. However, the parameter names can differ.

A definition is also a declaration, because the full definition of an entity also tells C++ how to use that entity.

The distinction between declaration and definition is crucial in C++. So far, our simple programs have not needed to face the difference, but that will soon change. Remember: A declaration describes a name to the compiler, and a definition provides all the details the compiler requires for the entity you are defining.

In order to use a variable, such as a function parameter, the compiler needs only the declaration of its name and the type. For a local variable, however, the compiler needs a definition, so that it knows to set aside memory to store the variable. The definition can also provide the variable's initial value. Even without an explicit initial value, the compiler may generate code to initialize the variable, such as ensuring that a `string` or `vector` is properly initialized as empty.

Counting Words—Again

Your turn. **Rewrite the word-counting program (last seen in Exploration 19), this time making use of functions**. For example, you can restore the pretty-printing utility by encapsulating it in a single function. Here's a hint: you may want to use the `typedef` names in multiple functions. If so, declare them before the first function, following the `#include` directives.

Test the program to ensure that your changes have not altered its behavior.

Compare your program with Listing 20-3.

Listing 20-3. Using Functions to Clarify the Word-Counting Program

```
#include <iomanip>
#include <iostream>
#include <locale>
#include <map>
#include <string>
```

```
typedef std::map<std::string, int> count_map;  ///< Map words to counts
typedef count_map::value_type      count_pair; ///< pair of a word and a count
typedef std::string::size_type     str_size;   ///< String size type

/** Initialize the I/O streams by imbuing them with
 * the given locale. Use this function to imbue the streams
 * with the native locale. C++ initially imbues streams with
 * the classic locale.
 * @param locale the native locale
 */
void initialize_streams(std::locale locale)
{
  std::cin.imbue(locale);
  std::cout.imbue(locale);
}

/** Find the longest key in a map.
 * @param map the map to search
 * @returns the size of the longest key in @p map
 */
str_size get_longest_key(count_map map)
{
  str_size result{0};
  for (auto pair : map)
    if (pair.first.size() > result)
      result = pair.first.size();
  return result;
}

/** Print the word, count, newline. Keep the columns neatly aligned.
 * Rather than the tedious operation of measuring the magnitude of all
 * the counts and then determining the necessary number of columns, just
 * use a sufficiently large value for the counts column.
 * @param iter an iterator that points to the word/count pair
 * @param longest the size of the longest key; pad all keys to this size
 */
void print_pair(count_pair pair, str_size longest)
{
  const int count_size{10}; // Number of places for printing the count
  std::cout << std::setw(longest)    << std::left  << pair.first <<
               std::setw(count_size) << std::right << pair.second << '\n';
}

/** Print the results in neat columns.
 * @param counts the map of all the counts
 */
```

```
void print_counts(count_map counts)
{
  str_size longest(get_longest_key(counts));

  // For each word/count pair...
  for (count_pair pair : counts)
    print_pair(pair, longest);
}

/** Sanitize a string by keeping only alphabetic characters.
 * @param str the original string
 * @param loc the locale used to test the characters
 * @return a santized copy of the string
 */
std::string sanitize(std::string str, std::locale loc)
{
  std::string result{};
  for (char ch : str)
    if (std::isalnum(ch, loc))
      result.push_back(std::tolower(ch, loc));
  return result;
}

/** Main program to count unique words in the standard input. */
int main()
{
  std::locale native{""};             // get the native locale
  initialize_streams(native);

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
  {
    std::string copy{ sanitize(word, native) };

    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }

  print_counts(counts);
}
```

By using functions, you can read, write, and maintain a program in smaller pieces, handling each piece as a discrete entity. Instead of being overwhelmed by one long `main()`, you can read, understand, and internalize one function at a time, then move on to the next function. The compiler keeps you honest by ensuring that the function calls match the function declarations, that the function definitions and declarations agree, that you haven't mistyped a name, and that the function return types match the contexts where the functions are called.

The main() Function

Now that you know more about functions, you can answer the question that you may have already asked yourself: **What is special about the `main()` function?**

____

____

One way that `main()` differs from ordinary functions is immediately obvious. All the `main()` functions in this book lack a `return` statement. An ordinary function that returns an `int` must have at least one `return` statement, but `main()` is special. If you don't supply your own `return` statement, the compiler inserts a `return 0;` statement at the end of `main()`. If control reaches the end of the function body, the effect is the same as `return 0;`, which returns a success status to the operating system. If you want to signal an error to the operating system, you can return a non-zero value from `main()`. How the operating system interprets the value depends on the implementation. The only portable values to return are `0`, `EXIT_SUCCESS`, and `EXIT_FAILURE. EXIT_SUCCESS` means the same thing as `0`—namely, success, but its actual value can be different from `0`. The names are declared in `<cstdlib>`.

The next Exploration continues to examine functions by taking a closer look at the arguments in function calls.

EXPLORATION 21

Function Arguments

This Exploration continues the examination of functions introduced in Exploration 20, by focusing on argument-passing. Take a closer look. Remember that *arguments* are the expressions that you pass to a function in a function call. *Parameters* are the variables that you declare in the function declaration. This Exploration introduces the topic of function arguments, an area of C++ 11 that is surprisingly complex and subtle.

Argument Passing

Read through Listing 21-1 then answer the questions that follow it.

Listing 21-1. Function Arguments and Parameters

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

void modify(int x)
{
  x = 10;
}

int triple(int x)
{
  return 3 * x;
}

void print_vector(std::vector<int> v)
{
  std::cout << "{ ";
  std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << "}\n";
}

void add(std::vector<int> v, int a)
{
  for (auto iter(v.begin()), end(v.end()); iter != end; ++iter)
    *iter = *iter + a;
}
```

```
int main()
{
  int a{42};
  modify(a);
  std::cout << "a=" << a << '\n';

  int b{triple(14)};
  std::cout << "b=" << b << '\n';

  std::vector<int> data{ 10, 20, 30, 40 };

  print_vector(data);
  add(data, 42);
  print_vector(data);
}
```

Predict what the program will print.

Now compile and run the program. **What does it actually print?**

Were you correct? ________________ **Explain why the program behaves as it does.**

When I run the program, I get the following results:

```
a=42
b=42
{ 10 20 30 40 }
{ 10 20 30 40 }
```

Expanding on these results, you may have noticed the `modify` function does not actually modify the variable `a` in `main()`, and the `add` function does not modify `data`. Your compiler might even have issued warnings to that effect.

As you can see, C++ passes arguments *by value*—that is, it copies the argument value to the parameter. The function can do whatever it wants with the parameter, but when the function returns, the parameter goes away, and the caller never sees any changes the function made.

If you want to return a value to the caller, use a `return` statement, as was done in the `triple` function.

Rewrite the add function so it returns the modified vector to the caller.

____

____

____

____

____

____

____

____

Compare your solution with the following code block:

```
std::vector<int> add(std::vector<int> v, int a)
{
  std::vector<int> result{};
  for (auto i : v)
    result.push_back(i + a);
  return result;
}
```

To call the new add, you must assign the function's result to a variable.

```
data = add(data, 42);
```

What is the problem with this new version of add?

____

____

Consider what would happen when you call add with a very large vector. The function makes an entirely new copy of its argument, consuming twice as much memory as it really ought to.

Pass-by-Reference

Instead of passing large objects (such as vectors) by value, C++ lets you pass them *by reference*. Add an ampersand (&) after the type name in the function parameter declaration. **Change Listing 21-1 to pass vector parameters by reference.** Also change the `modify` function, but leave the other `int` parameters alone. **What do you predict will be the output?**

____

____

____

____

Now compile and run the program. **What does it actually print?**

____

____

____

____

Were you correct? ________________ **Explain why the program behaves as it does.**

____

____

Listing 21-2 shows the new version of the program.

Listing 21-2. Pass Parameters by Reference

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

void modify(int& x)
{
  x = 10;
}

int triple(int x)
{
  return 3 * x;
}

void print_vector(std::vector<int>& v)
{
  std::cout << "{ ";
  std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << "}\n";
}

void add(std::vector<int>& v, int a)
{
  for (auto iter(v.begin()), end(v.end()); iter != end; ++iter)
    *iter = *iter + a;
}

int main()
{
  int a{42};
  modify(a);
  std::cout << "a=" << a << '\n';
  int b{triple(14)};
  std::cout << "b=" << b << '\n';

  std::vector<int> data{ 10, 20, 30, 40 };

  print_vector(data);
  add(data, 42);
  print_vector(data);
}
```

When I run the program, I get the following results:

```
a=10
b=42
{ 10 20 30 40 }
{ 52 62 72 82 }
```

This time, the program modified the `x` parameter in `modify` and updated the vector's contents in `add`. **Change the rest of the parameters to use pass-by-reference. What do you expect to happen?**

____

____

Try it. **What actually happens?**

____

____

The compiler does not allow you to call `triple(14)` when `triple`'s parameter is a reference. Consider what would happen if `triple` attempted to modify its parameter. You can't assign to a number, only to a variable. Variables and literals fall into different categories of expressions. In general terms, a variable is an *lvalue*, as are references. A literal is called an *rvalue*, and expressions that are built up from operators and function calls usually result in rvalues. When a parameter is a reference, the argument in the function call must be an lvalue. If the parameter is call-by-value, you can pass an rvalue.

Can you pass an lvalue to a call-by-value parameter? ________________

You've seen many examples that you can pass an lvalue. C++ automatically converts any lvalue to an rvalue when it needs to. **Can you convert an rvalue to an lvalue?** ________________

If you aren't sure, try to think of the problem in more concrete terms: can you convert an integer literal to a variable? That means you cannot convert an rvalue to an lvalue. Except, sometimes you can, as the next section will explain.

const References

In the modified program, the `print_vector` function takes its parameter by reference, but it doesn't modify the parameter. This opens a window for programming errors: you can accidentally write code to modify the vector. To prevent such errors, you can revert to call-by-value, but you would still have a memory problem if the argument is large. Ideally, you would be able to pass an argument by reference, but still prevent the function from modifying its parameter. Well, as it turns out, such a method does exist. Remember `const`? C++ lets you declare a function parameter `const` too.

```
void print_vector(std::vector<int> const& v)
{
  std::cout << "{ ";
  std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << "}\n";
}
```

Read the parameter declaration by starting at the parameter name and working your way from right to left. The parameter name is `v`; it is a reference; the reference is to a `const` object; and the object type is `std::vector<int>`. Sometimes, C++ can be hard to read, especially for a newcomer to the language, but with practice, you will soon read such declarations with ease.

CONST WARS

Many C++ programmers put the `const` keyword in front of the type, as demonstrated in the following:

```
void print_vector(const std::vector<int>& v)
```

For simple definitions, the `const` placement is not critical. For example, to define a named constant, you might use

```
const int max_width{80}; // maximum line width before wrapping
```

The difference between that and

```
int const max_width{80}; // maximum line width before wrapping
```

is small. But with a more complicated declaration, such as the parameter to `print_vector`, the different style is more significant. I find my technique much easier to read and understand. My rule of thumb is to keep the `const` keyword as close as possible to whatever it is modifying.

More and more C++ programmers are coming around to adopt the `const`-near-the-name style instead of `const` out in front. Again, this is an opportunity for you to be in the vanguard of the most up-to-date C++ programming trends. But you have to get used to reading code with `const` out in front, because you're going to see a lot of it.

So, `v` is a reference to a `const` vector. Because the vector is `const`, the compiler prevents the `print_vector` function from modifying it (adding elements, erasing elements, changing elements, and so on). Go ahead and try it. See what happens if you throw in any one of the following lines:

```
v.push_back(10); // add an element
v.pop_back();    // remove the last element
v.front() = 42;  // modify an element
```

The compiler stops you from modifying a `const` parameter.

Standard practice is to use references to pass any large data structure, such as `vector`, `map`, or `string`. If the function has no intention of making changes, declare the reference as a `const`. For small objects, such as `int`, use pass-by-value.

If a parameter is a reference to `const`, you can pass an rvalue as an argument. This is the exception that lets you convert an rvalue to an lvalue. To see how this works, change `triple`'s parameter to be a reference to `const`.

```
int triple(int const& x)
```

Convince yourself that you can pass an rvalue (such as 14) to `triple`. Thus, the more precise rule is that you can convert an rvalue to a `const` lvalue, but not to a non-`const` lvalue.

const_iterator

One additional trick you have to know when using `const` parameters: if you need an iterator, use `const_iterator` instead of `iterator`. A `const` variable of type `iterator` is not very useful, because you cannot modify its value, so the iterator cannot advance. You can still modify the element by assigning to the dereferenced iterator (e.g., `*iter`). Instead, a `const_iterator` can be modified and advanced, but when you dereference the iterator, the resulting object is `const`. Thus, you can read values but not modify them. This means you can safely use a `const_iterator` to iterate over a `const` container.

```
void print_vector(std::vector<int> const& v)
{
  std::cout << "{ ";
  std::string separator{};
  for (std::vector<int>::const_iterator i{v.begin()}, end{v.end()}; i != end; ++i)
  {
    std::cout << separator << *i;
    separator = ", ";

  }
  std::cout << "}\n";
}
```

You can do the same with a range-based for loop, but I wanted to show the use of `const_iterator`.

Multiple Output Parameters

You've already seen how to return a value from a function. And you've seen how a function can modify an argument by declaring the parameter as a reference. You can use reference parameters to "return" multiple values from a function. For example, you may want to write a function that reads a pair of numbers from the standard input, as shown in the following:

```
void read_numbers(int& x, int& y)
{
  std::cin >> x >> y;
}
```

Now that you know how to pass strings, vectors, and whatnot to a function, you can begin to make further improvements to the word-counting program, as you will see in the next Exploration.

EXPLORATION 22

Using Algorithms

So far, your use of the standard algorithms has been limited to a few calls to copy, the occasional use of sort, and so on. The main limitation has been that many of the more interesting algorithms require you to supply a function in order to do anything useful. This Exploration takes a look at these more advanced algorithms. In addition, we'll revisit some of the algorithms you already know, to show you how they, too, can be used in a more advanced manner.

Transforming Data

Several programs that you've read and written have a common theme: copying a sequence of data, such as a vector or string, and applying some kind of transformation to each element (converting to lowercase, doubling the values in an array, and so on). The standard algorithm, transform, is ideal for applying an arbitrarily complex transformation to the elements of a sequence.

For example, recall Listing 10-5, which doubled all the values in an array. Listing 22-1 presents a new way to write this same program, but using transform.

Listing 22-1. Calling transform to Apply a Function to Each Element of an Array

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int times_two(int i)
{
  return i * 2;
}

int main()
{
   std::vector<int> data{};

   std::copy(std::istream_iterator<int>(std::cin),
             std::istream_iterator<int>(),
             std::back_inserter(data));

   std::transform(data.begin(), data.end(), data.begin(), times_two);
   std::copy(data.begin(), data.end(),
             std::ostream_iterator<int>(std::cout, "\n"));
}
```

The transform function takes four arguments: the first two specify the input range (as start and one-past-the-end iterators), the third argument is a write iterator, and the final argument is the name of a function. Like other algorithms, transform is declared in the <algorithm> header.

Regarding the third argument, as usual, it is your responsibility to ensure the output sequence has enough room to accommodate the transformed data. In this case, the transformed data overwrite the original data, so the start of the output range is the same as the start of the input range. The fourth argument is just the name of a function that you must have declared or defined earlier in the source file. In this example, the function takes one int parameter and returns an int. The general rule for a transform function is that its parameter type must match the input type, which is the type of the element to which the read iterators refer. The return value must match the output type, which is the type to which the result iterator refers. The transform algorithm calls this function once for each element in the input range. It copies to the output range the value returned by the function.

Rewriting the word-counting program is a little harder. Recall from Listing 20-3 that the sanitize function transforms a string by removing non-letters and converting all uppercase letters to lowercase. The purpose of the C++ standard library is not to provide a zillion functions that cover all possible programming scenarios but, rather, to provide the tools you need to build your own functions with which you can solve your problems. Thus, you would search the standard library in vain for a single algorithm that copies, transforms, and filters. Instead, you can combine two standard functions: one that transforms and one that filters.

A further complication, however, is that you know that the filtering and transforming functions will rely on a locale. Solve the problem for now by setting your chosen locale as the global locale. Do this by calling std::local::global, and passing a locale object as the sole argument. An std::locale object created with the default constructor uses the global locale, so after your program sets your chosen locale as the global locale, you can easily imbue a stream or otherwise access the chosen locale by means of std::locale{}. Any function can use the global locale without having to pass locale objects around. Listing 22-2 demonstrates how to rewrite Listing 20-3 to set the global locale to the native locale and then how to use the global locale in the rest of the program.

Listing 22-2. New main Function That Sets the Global Locale

```
#include <iomanip>
#include <iostream>
#include <locale>
#include <map>
#include <string>

typedef std::map<std::string, int> count_map;  ///< Map words to counts
typedef count_map::value_type      count_pair; ///< pair of a word and a count
typedef std::string::size_type     str_size;   ///< String size type

/** Initialize the I/O streams by imbuing them with
 * the global locale. Use this function to imbue the streams
 * with the native locale. C++ initially imbues streams with
 * the classic locale.
 */
void initialize_streams()
{
  std::cin.imbue(std::locale{});
  std::cout.imbue(std::locale{});
}
```

```
/** Find the longest key in a map.
 * @param map the map to search
 * @returns the size of the longest key in @p map
 */
str_size get_longest_key(count_map const& map)
{
  str_size result{0};
  for (auto pair : map)
    if (pair.first.size() > result)
      result = pair.first.size();
  return result;
}

/** Print the word, count, newline. Keep the columns neatly aligned.
 * Rather than the tedious operation of measuring the magnitude of all
 * the counts and then determining the necessary number of columns, just
 * use a sufficiently large value for the counts column.
 * @param pair a word/count pair
 * @param longest the size of the longest key; pad all keys to this size
 */
void print_pair(count_pair const& pair, str_size longest)
{
  int const count_size{10}; // Number of places for printing the count
  std::cout << std::setw(longest)    << std::left  << pair.first <<
               std::setw(count_size) << std::right << pair.second << '\n';
}

/** Print the results in neat columns.
 * @param counts the map of all the counts
 */
void print_counts(count_map counts)
{
  str_size longest{get_longest_key(counts)};

  // For each word/count pair...
  for (count_pair pair: counts)
    print_pair(pair, longest);
}

/** Sanitize a string by keeping only alphabetic characters.
 * @param str the original string
 * @return a santized copy of the string
 */
std::string sanitize(std::string const& str)
{
  std::string result{};
  for (char c : str)
    if (std::isalnum(c, std::locale{}))
```

```
      result.push_back(std::tolower(c, std::locale{}));
  return result;
}

/** Main program to count unique words in the standard input. */
int main()
{
  // Set the global locale to the native locale.
  std::locale::global(std::locale{""});
  initialize_streams();

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  while (std::cin >> word)
  {
    std::string copy{sanitize(word)};

    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }

  print_counts(counts);
}
```

Now it's time to rewrite the `sanitize` function to take advantage of algorithms. Use `transform` to convert characters to lowercase. Use `remove_if` to get rid of nonalphabetic characters from the string. The `remove_if` algorithm calls a function for each element of a sequence. If the function returns true, `remove_if` eliminates that element from the sequence—well, it kind of does.

One curious side effect of how iterators work in C++ is that the `remove_if` function does not actually erase anything from the sequence. Instead, it rearranges the elements and returns an iterator that points to the position one past the end of the elements to be preserved. You can then call the `erase` member function to delete the elements that were removed, or you can make sure your function keeps track of the new logical end of the sequence.

Figure 22-1 illustrates how `remove_if` works with a *before* and *after* view of a string. Notice how the `remove_if` function does not alter the size of the string, but the characters after the new end are not meaningful. They are junk.

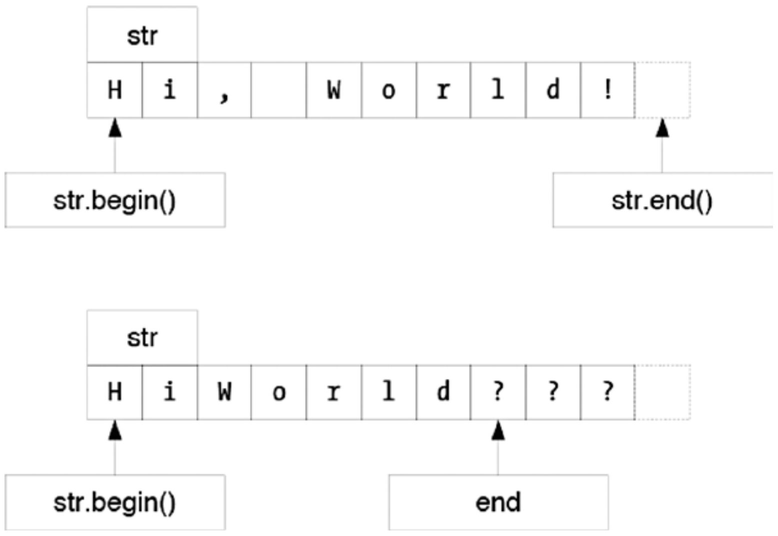


Figure 22-1. *Removing elements from a sequence*

Take a look at Listing 22-3 to see how `remove_if` works in code.

Listing 22-3. Sanitizing a String by Transforming It

```
/** Test for non-letter.
 * @param ch the character to test
 * @return true if @p ch is not a character that makes up a word
 */
bool non_letter(char ch)
{
  return not std::isalnum(ch, std::locale());
}

/** Convert to lowercase.
 * Use a canonical form by converting to uppercase first,
 * and then to lowercase.
 * @param ch the character to test
 * @return the character converted to lowercase
 */
char lowercase(char ch)
{
  return std::tolower(ch, std::locale());
}

/** Sanitize a string by keeping only alphabetic characters.
 * @param str the original string
 * @return a santized copy of the string
 */
std::string sanitize(std::string str)
```

```
{
  // Remove all non-letters from the string, and then erase them.
  str.erase(std::remove_if(str.begin(), str.end(), non_letter),
            str.end());

  // Convert the remnants of the string to lowercase.
  std::transform(str.begin(), str.end(), str.begin(), lowercase);

  return str;
}
```

The `erase` member function takes two iterators as arguments and erases all the elements within that range. The `remove_if` function returns an iterator that points to one-past-the-end of the new `string`, which means it also points to the first position of the elements to be erased. Passing `str.end()` as the end of the range instructs `erase` to get rid of all the removed elements.

The remove/erase idiom is common in C++, so you should get used to seeing it. The standard library has several remove-like functions, all of which work the same way. It takes a little while to get used to this approach, but once you do, you will find it quite easy to use.

Predicates

The `non_letter` function is an example of a *predicate*. A predicate is a function that returns a `bool` result. These functions have many uses in the standard library.

For example, the `sort` function sorts values in ascending order. What if you wanted to sort data in descending order? The `sort` function lets you provide a predicate to compare items. The ordering predicate (call it `pred`) must meet the following qualifications:

- `pred(a, a)` must be `false` (a common error is to implement `<=` instead of `<`, which violates this requirement).
- If `pred(a, b)` is `true`, and `pred(b, c)` is `true`, then `pred(a, c)` must also be `true.`
- The parameter types must match the element type to be sorted.
- The return type must be `bool` or something that C++ can convert automatically to `bool.`

If you don't provide a predicate, `sort` uses the `<` operator as the default.

Write a predicate to compare two integers for sorting in descending order.

____

____

____

____

____

____

Write a program to test your function. **Did it work?** ________________

Compare your solution with Listing 22-4.

Listing 22-4. Sorting into Descending Order

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

/** Predicate for sorting into descending order. */
int descending(int a, int b)
{
  return a > b;
}

int main()
{
  std::vector<int> data{};
  data.insert(data.begin(), std::istream_iterator<int>(std::cin),
                            std::istream_iterator<int>());

  std::sort(data.begin(), data.end(), descending);

  std::copy(data.begin(), data.end(), std::ostream_iterator<int>(std::cout, "\n"));
}
```

The default comparison that `sort` uses (the `<` operator) is the standard for comparison throughout the standard library. The standard library uses `<` as the ordering function for anything and everything that can be ordered. For example, `map` uses `<` to compare keys. The `lower_bound` functions (which you used in Exploration 13) use the `<` operator to perform a binary search.

The standard library even uses `<` to compare objects for equality when dealing with ordered values, such as a map or a binary search. (Algorithms and containers that are not inherently ordered use `==` to determine when two objects are equal.) To test if two items, `a` and `b`, are the same, these library functions use `a < b` and `b < a`. If both comparisons are `false`, then `a` and `b` must be the same, or in C++ terms, *equivalent*. If you supply a comparison predicate (`pred`), the library considers `a` and `b` to be equivalent if `pred(a, b)` is `false` and `pred(b, a)` is `false`.

Modify your descending-sort program (or Listing 22-4) to use `==` as the comparison operator. What do you expect to happen?

____

____

Run the new program with a variety of inputs. **What actually happens?**

____

____

Were you correct? ______________

The equivalence test is broken, because `descending(a,a)` is `true`, not `false`. Because the predicate does not work properly, `sort` is not guaranteed to work properly, or at all. The results are undefined. Whenever you write a predicate, be sure the comparison is strict (that is, you can write a valid equivalence test) and the transitive property holds (if `a < b` and `b < c`, then `a < c` is also `true`).

Other Algorithms

The standard library contains too many useful algorithms to cover in this book, but I'll take a few moments in this section to introduce you to at least some of them. Refer to a comprehensive language reference to learn about the other algorithms.

Let's explore algorithms by looking for palindromes. A palindrome is a word or phrase that reads the same forward and backward, ignoring punctuation, such as, for example:

Madam, I'm Adam.

The program reads one line of text at a time by calling the `getline` function. This function reads from an input stream into a string, stopping when it reads a delimiter character. The default delimiter is `'\n'`, so it reads one line of text. It does not skip over initial or trailing white space.

The first step is to remove non-letter characters, but you already know how to do that.

The next step is to test whether the resulting string is a palindrome. The `reverse` function transposes the order of elements in a range, such as characters in a string.

The `equal` function compares two sequences to determine whether they are the same. It takes two iterators for the first range and a starting iterator for the second range. It assumes the two ranges are the same size, and it compares one element at a time and works for any kind of sequence. In this case, the comparison must be case-insensitive, so provide a predicate that converts all text to a canonical case and then compares them.

Go ahead. **Write the program.** A simple web search should deliver up some juicy palindromes with which to test your program. If you don't have access to the Web, try the following:

Eve

Deed

Hannah

Leon saw I was Noel

If you need some hints, here are my recommendations:

- Write a function called `is_palindrome` that takes a string as a parameter and returns a `bool`.
- This function uses the `remove_if` function to clean up the string.
- There's no need to erase the removed elements, however. Instead, save the end iterator returned by `remove_if` and use that instead of `str.end()`.
- Define a new string, `rev`, which is a copy of the sanitized string. Copy only the desired part of the test string by taking only up to the iterator that was returned from `remove_if`.
- Next, call `reverse`.
- Write a predicate, `is_same_char`, to compare two characters after converting them both to uppercase and then to lowercase.
- Call the `equal` function with `same_char` as its predicate, to compare the original string with the reversed copy.
- The `main` program sets the global locale to the native locale and imbues the input and output streams with the new global locale.
- The main program calls `getline(std::cin, line)` until the function returns `false` (meaning error or end-of-file), then calls `is_palindrome` for each line.

Listing 22-5 shows my version of the completed program.

Listing 22-5. Testing for Palindromes

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <locale>
#include <string>

/** Test for non-letter.
 * @param ch the character to test
 * @return true if @p ch is not a letter
 */
bool non_letter(char ch)
{
  return not std::isalpha(ch, std::locale());
}

/** Convert to lowercase.
 * Use a canonical form by converting to uppercase first,
 * and then to lowercase.
 * @param ch the character to test
 * @return the character converted to lowercase
 */
char lowercase(char ch)
{
  return std::tolower(ch, std::locale());
}

/** Compare two characters without regard to case. */
bool is_same_char(char a, char b)
{
  return lowercase(a) == lowercase(b);
}

/** Determine whether @p str is a palindrome.
 * Only letter characters are tested. Spaces and punctuation don't count.
 * Empty strings are not palindromes because that's just too easy.
 * @param str the string to test
 * @return true if @p str is the same forward and backward
 */
bool is_palindrome(std::string str)

{
  std::string::iterator end{std::remove_if(str.begin(), str.end(), non_letter)};
  std::string rev{str.begin(), end};
  std::reverse(rev.begin(), rev.end());
  return not rev.empty() and std::equal(str.begin(), end, rev.begin(), is_same_char);
}
```

```
int main()
{
  std::locale::global(std::locale{""});
  std::cin.imbue(std::locale{});
  std::cout.imbue(std::locale{});

  std::string line{};
  while (std::getline(std::cin, line))
    if (is_palindrome(line))
      std::cout << line << '\n';
}
```

In a large program, the predicate or transformation function might be declared far from where it is used. Often, a predicate is used only once. By defining a function just for that predicate, it makes your program harder to understand. The human reader must read all the code to ensure that the predicate truly is called in only one place. It would be nice if C++ offered a way to write the predicate in the place where it is used, and thereby avoid these problems. Read the next Exploration to learn how to achieve this in C++ 11.

EXPLORATION 23

Unnamed Functions

One problem with calling algorithms is that sometimes the predicate or transformation function must be declared far from the place where it is called. With a properly descriptive name, this problem can be reduced, but often the function is trivial, and your program would be much easier to read if you could put its functionality directly in the call to the standard algorithm. A new feature in C++ 11 permits exactly that.

Lambdas

C++ 11 lets you define a function as an expression. You can pass this function to an algorithm, save it in a variable, or call it immediately. Such a function is called a *lambda,* for reasons that only a computer scientist would understand or even care about. If you aren't a computer scientist, don't worry, and just realize that when the nerds talk about lambdas, they are just talking about unnamed functions. As a quick introduction, Listing 23-1 rewrites Listing 22-1 to use a lambda.

Listing 23-1. Calling `transform` to Apply a Lambda to Each Element of an Array

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
   std::vector<int> data{};

   std::copy(std::istream_iterator<int>(std::cin),
             std::istream_iterator<int>(),
             std::back_inserter(data));

   std::transform(data.begin(), data.end(), data.begin(), [](int x) { return x * 2; });

   std::copy(data.begin(), data.end(),
             std::ostream_iterator<int>(std::cout, "\n"));
}
```

The lambda almost looks like a function definition. Instead of a function name, a lambda begins with square brackets. The usual function arguments and compound statement follow. **What's missing?**

____

That's right, the function's return type. When a lambda's function body contains only a single return statement, the compiler deduces the function's type from the type of the return expression. In this case, the return type is `int`.

With a lambda, the program is slightly shorter and much easier to read. You don't have to hunt for the definition of `times_two()` to learn what it does. (Not all functions are named so clearly.) But lambdas are even more powerful and can do things that ordinary functions can't. Take a look at Listing 23-2 to see what I mean.

Listing 23-2. Using a Lambda to Access Local Variables

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
   std::vector<int> data{};

   std::cout << "Multiplier: ";
   int multiplier{};
   std::cin >> multiplier;
   std::cout << "Data:\n";
   std::copy(std::istream_iterator<int>(std::cin),
      std::istream_iterator<int>(),
      std::back_inserter(data));

   std::transform(data.begin(), data.end(), data.begin(),
      [multiplier](int i){ return i * multiplier; });

   std::copy(data.begin(), data.end(),
      std::ostream_iterator<int>(std::cout, "\n"));
}
```

Predict the output if the program's input is as follows:

```
4 1 2 3 4 5
```

____

____

____

____

The first number is the multiplier, and the remaining numbers are multiplied by it, yielding

```
4
8
12
16
20
```

See the trick? The lambda is able to read the local variable, `multiplier`. A separate function, such as `times_two()` in Listing 22-1, can't do that. Of course, you can pass two arguments to `times_two()`, but this use of the `transform` algorithm calls the function with only one argument. There are ways to work around this limitation, but I won't bother showing them to you, because lambdas solve the problem simply and elegantly.

Naming an Unnamed Function

Although a lambda is an unnamed function, you can give it a name by assigning the lambda to a variable. This is a case where you probably want to declare the variable using the `auto` keyword, so you don't have to think about the type of thelambda that is the variable's initial value. Just remember that `auto` doesn't play well with universal initialization, so use an equal sign or parentheses, as in the following example:

```
auto times_three = [](int i) { return i * 3; };
```

Once you assign a lambda to a variable, you can call that variable as though it were an ordinary function:

```
int forty_two{ times_three(14) };
```

The advantage of naming a lambda is that you can call it more than once in the same function. In this way, you get the benefit of self-documenting code with a well-chosen name and the benefit of a local definition.

If you don't want to use `auto`, the standard library can help. In the `<functional>` header is the type `std::function`, which you use by placing a function's return type and parameter types in angle brackets, e.g., `std::function<int(int)>`. For example, the following defines a variable, `times_two`, and initializes it with a lambda that takes one argument of type `int` and returns `int`:

```
std::function<int(int)> times_two{ [](int i) { return i * 2; } };
```

The actual type of a lambda is more complicated, but the compiler knows how to convert that type to the matching `std::function<>` type.

Capturing Local Variables

Naming a local variable in the lambda's square brackets is called *capturing* the variable. If you do not capture a variable, you cannot use it in the lambda, so the lambda would be able to use only its function parameters.

Read the program in Listing 23-3. Think about how it captures the local variable, `multiplier`.

Listing 23-3. Using a Lambda to Access Local Variables

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
   std::vector<int> data{ 1, 2, 3 };

   int multiplier{3};
   auto times = [multiplier](int i) { return i * multiplier; };

   std::transform(data.begin(), data.end(), data.begin(), times);

   multiplier = 20;
   std::transform(data.begin(), data.end(), data.begin(), times);

   std::copy(data.begin(), data.end(),
             std::ostream_iterator<int>(std::cout, "\n"));
}
```

Predict the output from Listing 23-3.

____

____

____

____

____

____

Now run the program. Was your prediction correct? ________________ **Why or why not?**

____

____

The value of `multiplier` was captured by value when the lambda was defined. Thus, changing the value of `multiplier` later does not change the lambda, and it still multiplies by three. The `transform()` algorithm is called twice, so the effect is to multiply by 9, not 60.

If you need your lambda to keep track of the local variable and always use its most recent value, you can capture a variable by reference by prefacing its name with an ampersand (similar to a reference function parameter), as in the following example:

```
[&multiplier](int i) { return i * multiplier; };
```

Modify Listing 23-3 to capture `multiplier` by reference. Run the program to observe its new behavior.

You can choose to omit the capture name to capture all local variables. Use an equal sign to capture everything by value or just an ampersand to capture everything by reference.

```
int x{0}, y{1}, z{2};
auto capture_all_by_value = [=]() { return x + y + z; };
auto capture_all_by_reference = [&]() { x = y = z = 0; };
```

I advise against capturing everything by default, because it leads to sloppy code. Be explicit about the variables that the lambda captures. The list of captures should be short, or else you are probably doing something wrong. Nonetheless, you will likely see other programmers capture everything, if only out of laziness, so I had to show you the syntax.

If you follow best practices and list individual capture names, the default is capture-by-value, so you must supply an ampersand for each name that you want to capture by reference. Feel free to mix capture-by-value and capture-by-reference.

```
auto lambda =
   [by_value, &by_reference, another_by_value, &another_by_reference]() {
      by_reference = by_value;
      another_by_reference = another_by_value;
   };
```

const Capture

Capture-by-value has one trick up its sleeve that can take you by surprise. Consider the simple program in Listing 23-4.

Listing 23-4. Using a Lambda to Access Local Variables

```
#include <iostream>

int main()
{
   int x{0};
   auto lambda = [x](int y) {
      x = 1;
      y = 2;
      return x + y;
   };
   int local{0};
   std::cout << lambda(local) << ", " << x << ", " << local << '\n';

}
```

What do you expect to happen when you run this program?

____**What is the surprise?**

____

You already know that function parameters are call-by-value, so the `y = 1` assignment has no effect outside of the lambda, and `local` remains 0. A by-value capture is similar in that you cannot change the local variable that is captured (`x` in Listing 23-4). But the compiler is even pickier than that. It doesn't let you write the assignment `x = 1`. It's as though every by-value capture were declared `const`.

Lambdas are different from ordinary functions in that default for by-value captures is `const`, and to get a non-`const` capture, you must explicitly tell the compiler. The keyword to use is `mutable`, which you put after the function parameters, as shown in Listing 23-5.

Listing 23-5. Using the mutable Keyword in a Lambda

```
#include <iostream>

int main()
{
   int x{0};
   auto lambda = [x](int y) mutable {
      x = 1;
      y = 2;
      return x + y;
   };
   int local{0};
   std::cout << lambda(local) << ", " << x << ", " << local << '\n';
}
```

Now the compiler lets you assign to the capture, x. The capture is still by-value, so x in main() doesn't change. The output of the program is

```
3, 0, 0
```

So far, I have never found an instance when I wanted to use mutable. It's there if you need it, but you will probably never need it.

Return Type

The return type of a lambda is the type of the return expression, if the lambda body contains only a return statement. But what if the lambda is more complicated? Although some argue that the compiler should be able to deduce the type from a more complicated lambda (with the restriction that every return statement have the same type), that is not what the standard says. Future updates to the standard will loosen this restriction, but as I write this, the restriction remains in place.

But the syntax for a lambda does not lend itself to declaring a function return type in the usual way. Instead, the return type follows the function parameter list, with an arrow (->) between the closing parenthesis and the return type:

```
[](int i) -> int { return i * 2; }
```

In general, the lambda is easier to read without the explicit return type. The return type is usually obvious, but if it is not, go ahead and be explicit. Clarity trumps brevity.

Rewrite Listing 22-5 to take advantage of lambdas. Write functions where you think functions are appropriate, and write lambdas where you think lambdas are appropriate. Compare your solution with mine in Listing 23-6.

Listing 23-6. Testing for Palindromes

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <locale>
#include <string>

/** Determine whether @p str is a palindrome.
 * Only letter characters are tested. Spaces and punctuation don't count.
 * Empty strings are not palindromes because that's just too easy.
 * @param str the string to test
 * @return true if @p str is the same forward and backward
 */
bool is_palindrome(std::string str)
{
  // Filter the string to keep only letters
  std::string::iterator end{std::remove_if(str.begin(), str.end(),
    [](char ch)
    {
      return not std::isalpha(ch, std::locale());
    })
  };
  // Reverse the filtered string.
  std::string rev{str.begin(), end};
  std::reverse(rev.begin(), rev.end());

  // Compare the filtered string with its reversal, without regard to case.
  return not rev.empty() and std::equal(str.begin(), end, rev.begin(),
    [](char a, char b)
    {
        auto lowercase = [](char ch)
        {
           return std::tolower(ch, std::locale());
        };
        return lowercase(a) == lowercase(b);
    }
  );
}

int main()
{
  std::locale::global(std::locale{""});
  std::cin.imbue(std::locale{});
  std::cout.imbue(std::locale{});

  std::string line{};
  while (std::getline(std::cin, line))
    if (is_palindrome(line))
      std::cout << line << '\n';
}
```

You may have noticed that some algorithms have more than one form. The `sort` function, for example, can take two iterators as arguments, or it can take two iterators and a predicate. Using one name for more than one function is called *overloading*. This is the subject of the next Exploration.

EXPLORATION 24

Overloading Function Names

In C++, multiple functions can have the same name, provided the functions have a different number of arguments or different argument types. Using the same name for multiple functions is called *overloading* and is common in C++.

Overloading

All programming languages use overloading at one level or another. For example, most languages use + for integer addition as well as for floating-point addition. Some languages, such as Pascal, use different operators for integer division (`div`) and floating-point division (`/`), but others, such as C and Java, use the same operator (`/`).

C++ takes overloading one step further, letting you overload your own function names. Judicious use of overloading can greatly reduce complexity in a program and make your programs easier to read and understand.

For example, C++ inherits several functions from the standard C library that compute an absolute value: `abs` takes an `int` argument; `fabs` takes a floating-point argument; and `labs` takes a long integer argument.

Note Don't be concerned that I have not yet covered these other types. All that matters for the purpose of this discussion is that they are distinct from `int`. The next Exploration will begin to examine them more closely, so please be patient.

C++ also has its own `complex` type for complex numbers, which has its own absolute value function. In C++, however, they all have the same name, `std::abs`. Using different names for different types merely clutters the mental landscape and contributes nothing to the clarity of the code.

The `sort` function, just to name one example, has two overloaded forms:

```
std::sort(start, end);
std::sort(start, end, compare);
```

The first form sorts in ascending order, comparing elements with the `<` operator, and the second form compares elements by calling `compare`. Overloading appears in many other places in the standard library. For example, when you create a `locale` object, you can copy the global locale by passing no arguments

```
std::isalpha('X', std::locale{});
```

or create a native locale object by passing an empty string argument

```
std::isalpha('X', std::locale{""});
```

Overloading functions is easy, so why not jump in? **Write a set of functions, all named print.** They all have a `void` return type and take various parameters.

- One takes an `int` as a parameter. It prints the parameter to the standard output.
- Another takes two `int` parameters. It prints the first parameter to the standard output and uses the second parameter as the field width.
- Another takes a `vector<int>` as the first parameter, followed by three `string` parameters. Print the first `string` parameter, then each element of the `vector` (by calling `print`), with the second `string` parameter between elements, and the third `string` parameter after the `vector`. If the `vector` is empty, print the first and third `string` parameters only.
- Another has the same parameters as the `vector` form, but also takes an `int` as the field width for each `vector` element.

Write a program to print vectors using the print functions. Compare your functions and program with mine in Listing 24-1.

Listing 24-1. Printing Vectors by Using Overloaded Functions

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

void print(int i)
{
  std::cout << i;
}

void print(int i, int width)
{
  std::cout.width(width);
  std::cout << i;
}

void print(std::vector<int> const& vec,
    int width,
    std::string const& prefix,
    std::string const& separator,
    std::string const& postfix)
{
  std::cout << prefix;

  bool print_separator{false};
  for (auto x : vec)
  {
    if (print_separator)
```

```
      std::cout << separator;
    else
      print_separator = true;
    print(x, width);
  }

  std::cout << postfix;
}

void print(std::vector<int> const& vec,
    std::string const& prefix,
    std::string const& separator,
    std::string const& postfix)
{
  print(vec, 0, prefix, separator, postfix);
}

int main()
{
  std::vector<int> data{ 10, 20, 30, 40, 100, 1000, };

  std::cout << "columnar data:\n";
  print(data, 10, "", "\n", "\n");
  std::cout << "row data:\n";
  print(data, "{", ", ", "}\n");
}
```

The C++ library often uses overloading. For example, you can change the size of a `vector` by calling its `resize` member function. You can pass one or two arguments: the first argument is the new size of the `vector`. If you pass a second argument, it is a value to use for new elements, in case the new size is larger than the old size.

```
data.resize(10);      // if the old size < 10, use default of 0 for new elements
data.resize(20, -42); // if the old size < 20, use -42 for new elements
```

Library-writers often employ overloading, but applications programmers use it less often. **Practice writing libraries by writing the following functions:**

bool is_alpha(char ch)

Returns `true` if `ch` is an alphabetic character in the global locale; if not, returns `false`.

bool is_alpha(std::string const& str)

Returns `true`, if `str` contains only alphabetic characters in the global locale, or `false`, if any character is not alphabetic. Returns `true` if `str` is empty.

char to_lower(char ch)

Returns `ch` after converting it to lowercase, if possible; otherwise, returns `ch`. Use the global locale.

std::string to_lower(std::string str)

Returns a copy of `str` after converting its contents to lowercase, one character at a time. Copies verbatim any character that cannot be converted to lowercase.

char to_upper(char ch)

Returns `ch` after converting it to uppercase, if possible; otherwise, returns `ch`. Use the global locale.

std::string to_upper(std::string str)

Returns a copy of `str` after converting its contents to uppercase, one character at a time. Copies verbatim any character that cannot be converted to uppercase.

Compare your solution with mine, which is shown in Listing 24-2.

Listing 24-2. Overloading Functions in the Manner of a Library-Writer

```
#include <algorithm>
#include <iostream>
#include <locale>

bool is_alpha(char ch)
{
  return std::isalpha(ch, std::locale{});
}

bool is_alpha(std::string const& str)
{
  for (char ch : str)
    if (not is_alpha(ch))
      return false;
  return true;
}

char to_lower(char ch)
{
  return std::tolower(ch, std::locale{});
}

std::string to_lower(std::string str)
{
  for (char& ch : str)
    ch = to_lower(ch);
  return str;
}

char to_upper(char ch)
{
  return std::toupper(ch, std::locale{});
}
```

```
std::string to_upper(std::string str)
{
  for (char& ch : str)
    ch = to_upper(ch);
  return str;
}

int main()
{
  std::string str{};
  while (std::cin >> str)
  {
    if (is_alpha(str))
      std::cout << "alpha\n";
    else
      std::cout << "not alpha\n";
    std::cout << "lower: " << to_lower(str) << "\nupper: " << to_upper(str) << '\n';
  }
}
```

After waxing poetic about the usefulness of standard algorithms, such as `transform`, I turned around and wrote my own loops. If you tried to use a standard algorithm, I applaud you for your effort and apologize for tricking you.

If you want to pass an overloaded function to a standard algorithm, the compiler has to be able to tell which overloaded function you really mean. For some rather complicated reasons, the compiler has difficulty understanding situations such as the following:

```
std::string to_lower(std::string str)
{
  std::transform(str.begin(), str.end(), str.begin(), to_lower);

  return str;
}
```

Because `to_lower` is overloaded, the compiler does not know which `to_lower` you mean. C++ has ways to help the compiler understand what you mean, but it involves some nasty-looking code, and I'd rather you stay away from it for the time being. If you really insist on punishing yourself, look at code that works, but don't try to sprain your brain understanding it.

```
std::string to_lower(std::string str)
{
  std::transform(str.begin(), str.end(), str.begin(),
                 static_cast<char (*)(char)>(to_lower));
  return str;
}
```

If you look closely at `to_upper` and `to_lower`, you'll notice a couple of techniques that are different from other, similar functions. Can you spot them? **If so, what are they?**

____

____

____

____

The range-based for loops work with references! Like a pass-by-reference function parameter, the declaration

```
char& ch
```

is a reference to a character in the range. Thus, assigning to `ch` changes the character in the string. Note that `auto` can also declare a reference. If each element of the range is large, you should use references to avoid making unnecessary copies. Use a `const` reference if you do not have to modify the element, as follows:

```
for (auto const& big_item : container_full_of_big_things)
```

The other anomaly is that `to_lower` and `to_upper` string functions do not take `const` references but take plain `strings` as parameters. This means the argument is passed by value, which in turn means the compiler arranges to copy the string when passing the argument to the function. The function requires the copy, so this technique helps the compiler generate optimal code for copying the argument, and it saves you a step in writing the function. It's a small trick, but a useful one. This technique will be especially useful later in the book—so don't forget it.

The `is_alpha` string function does not modify its parameter, so it can take a `const` reference.

A common use of overloading is to overload a function for different types, including different integer types, such as a long integer. The next Exploration takes a look at these other types.

EXPLORATION 25

Big and Little Numbers

Another common use for overloading is to write functions that work just as well with large and small integers as with plain integers. C++ has five different integer types, ranging in size from 8 bits to 64 bits or larger, with several choices for sizes in between. This Exploration takes a look at the details.

The Long and Short of It

The size of an `int` is the natural size of an integer on the host platform. For your desktop computer, that probably means 32 bits or 64 bits. Not too long ago, it meant 16 bits or 32 bits. I've also used computers with 36-bit and 60-bit integers. In the realm of desktop computers and workstations, 32-bit and 64-bit processors dominate today's computing landscape, but don't forget specialized devices, such as digital signal processors (DSPs) and other embedded chips, where 16-bit architectures are still common. The purpose of leaving the standard flexible is to ensure maximum performance for your code. The C++ standard guarantees that an `int` can represent, at a minimum, any number in the range –32,767 to 32,767, inclusive.

Although your desktop computer most likely uses two's complement representation for integers, C++ does not mandate that format, only that the representation be binary. In other words, you should treat an integer as a number, not a bit pattern. (Wait for Exploration 64, if you need to work at the bit level.)

To discover the number of bits in an integer, use `std::numeric_limits`, as you did way back in Listing 2-3. **Try that same program, but substitute int for `bool`. What do you get for the output?** ________________

Most likely, you got 31, although some of you may have seen 15 or 63. The reason for this is that `digits` does not count the sign bit. No matter what representation your computer uses for an integer, one of those bits must indicate whether the number is negative or positive. Thus, for a type that represents a signed quantity, such as `int`, you must add one to `digits`, and for a type with no sign, such as `bool`, use `digits` without further modification. Fortunately, `std::numeric_limits` offers `is_signed`, which is `true` for a signed type and `false` for a type without a sign bit. **R**ewrite Listing 2-3 to use **is_signed to determine whether to add one to digits and print the number of bits per int and per `bool`.**

Check your answers. **Are they correct?** ________________ Compare your program with Listing 25-1.

Listing 25-1. Discovering the Number of Bits in an Integer

```
#include <iostream>
#include <limits>

int main()
{
  std::cout << "bits per int = ";
  if (std::numeric_limits<int>::is_signed)
    std::cout << std::numeric_limits<int>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<int>::digits << '\n';

  std::cout << "bits per bool = ";
  if (std::numeric_limits<bool>::is_signed)
    std::cout << std::numeric_limits<bool>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<bool>::digits << '\n';
}
```

Long Integers

Sometimes, you need more bits than `int` can handle. In this case, add `long` to the definition to get a long integer.

```
long int lots_o_bits{2147483647};
```

You can even drop the `int`, as shown in the following:

```
long lots_o_bits{2147483647};
```

The standard guarantees that a `long int` can handle numbers in the range –2,147,483,647 to 2,147,483,647, but an implementation can choose a large size. C++ does not guarantee that a `long int` is actually longer than a plain `int`. On some platforms, `int` might be 32 bits and `long` might be 64. When I first used a PC at home, an `int` was 16 bits and `long` was 32 bits. At times, I've used systems for which `int` and `long` were both 32 bits. I'm writing this book on a machine that uses 32 bits for `int` and 64 bits for `long`.

The type `long long int` can be even bigger, with a range of at least –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807. You can drop the `int`, if you wish, and programmers often do.

Use `long long` if you want to store numbers as large as possible and are willing to pay a small performance penalty (on some systems, or a large penalty on others). Use `long` if you have to ensure portability and must represent numbers outside the range ±32,767 (which is all C++ guarantees for type `int`).

Short Integers

Sometimes, you don't have the full range of an `int`, and reducing memory consumption is more important. In this case, use a `short int`, or just `short`, which has a guaranteed range of at least –32,767 to 32,767, inclusive. This is the same guaranteed range as `int`, but implementations often choose to make an `int` larger than the minimum and keep `short` in this range of 16 bits. But you must not assume that `int` is always larger than `short`. Both may be the exact same size.

As is done with `long`, you define a type as `short int` or `short`.

```
short int answer{42};
short zero{0};
```

Modify Listing 25-1 to print the number of bits in a `long` and a `short` too. How many bits are in a `long` on your system? _______________ How many in a `short`? _________________ `long long`? ____________

When I run the program in Listing 25-2, I get 16 bits in a `short`, 32 in an `int`, and 64 in a `long` and `long long`. On another computer in my network, I get 16 bits in a `short`, 32 in an `int` and `long`, and 64 in a `long long`.

Listing 25-2. Revealing the Number of Bits in Short and Long Integers

```
#include <iostream>
#include <limits>

int main()
{
  std::cout << "bits per int = ";
  if (std::numeric_limits<int>::is_signed)
    std::cout << std::numeric_limits<int>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<int>::digits << '\n';

  std::cout << "bits per bool = ";
  if (std::numeric_limits<bool>::is_signed)
    std::cout << std::numeric_limits<bool>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<bool>::digits << '\n';

  std::cout << "bits per short int = ";
  if (std::numeric_limits<short>::is_signed)
    std::cout << std::numeric_limits<short>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<short>::digits << '\n';

  std::cout << "bits per long int = ";
  if (std::numeric_limits<long>::is_signed)
    std::cout << std::numeric_limits<long>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<long>::digits << '\n';

  std::cout << "bits per long long int = ";
  if (std::numeric_limits<long long>::is_signed)
    std::cout << std::numeric_limits<long long>::digits + 1 << '\n';
  else
    std::cout << std::numeric_limits<long long>::digits << '\n';
}
```

Integer Literals

When you write an integer literal (that is, an integer constant), the type depends on its value. If the value fits in an `int`, the type is `int`; otherwise, the type is `long` or `long long`. You can force a literal to have type `long` by adding `l` or `L` (the letter *L* in lowercase or uppercase) after the digits. (Curiously, C++ has no way for you to type a `short` literal.) I always use uppercase `L` because a lowercase `l` looks too much like the digit `1`. The compiler can always tell the difference, but every year it gets a little harder for me to see the difference between `1` and `l`. Use two consecutive L's for a `long long`.

Devise a way for a program to print int=, followed by the value, for an `int` literal; print `long=`, followed by the value, for a `long` literal; and print `long long=`, followed by the value, for a `long long` literal. (Hint: What was the topic of the previous Exploration?) Write a program to demonstrate your idea, and test it with some literals. If you can, run the program on platforms that use different sizes for `int` and `long`. Compare your program to that of Listing 25-3.

Listing 25-3. Using Overloading to Distinguish Types of Integer Literals

```
#include <iostream>
#include <locale>

void print(int value)
{
  std::cout << "int=" << value << '\n';
}

void print(long value)
{
  std::cout << "long=" << value << '\n';
}

void print(long long value)
{
  std::cout << "long long=" << value << '\n';
}

int main()
{
  std::cout.imbue(std::locale{""});
  print(0);
  print(0L);
  print(32768);
  print(-32768);
  print(2147483647);
  print(-2147483647);
  print(2147483648);

  print(9223372036854775807);
  print(-9223372036854775807);
}
```

The actual types that your compiler chooses will vary. Your compiler may even be able to handle larger integer literals. The C++ standard sets down some guaranteed ranges for each of the integer types, so all the values in Listing 25-3 will work with a decent C++ 11 compiler. If you stick to the guaranteed ranges, your program will compile and run everywhere; outside the range, you're taking your chances. Library-writers have to be especially careful. You never know when someone working on a small, embedded processor might like your code and want to use it.

Byte-Sized Integers

The smallest integer type that C++ offers is `signed char`. The type name looks similar to the character type, `char`, but the type acts differently. It usually acts like an integer. By definition, the size of `signed char` is 1 byte, which is the smallest size that your C++ compiler supports for any type. The guaranteed range of `signed char` is –127 to 127.

In spite of the name, you should try not to think of `signed char` as a mutated character type; instead, think of it as a misspelled integer type. Many programs have a `typedef` similar to

```
typedef signed char byte;
```

to make it easier for you to think of this type as a byte-sized integer type.

There is no easy way to write a `signed char` literal, just as there is no way to write a simple `short` literal. Character literals have type `char`, not `signed char`. Besides, some characters may be out of range for `signed char`.

Although the compiler does its best to help you remember that `signed char` is not a `char`, the standard library is less helpful. The I/O stream types treat `signed char` values as characters. Somehow, you have to inform the stream that you want to print an integer, not a character. You also need a solution to create `signed char` (and `short`) literals. Fortunately, the same solution lets you use `signed char` constants and print `signed char` numbers: type casting.

Type Casting

Although you cannot write a `short` or arbitrary `signed char` literal directly, you can write a constant expression that has type `short` or `signed char` and take any suitable value. The trick is to use a plain `int` and tell the compiler exactly what type you want.

```
static_cast<signed char>(-1)
static_cast<short int>(42)
```

The expression does not have to be a literal, as demonstrated in the following:

```
int x{42};
static_cast<short>(x);
```

The `static_cast` expression is known as a *type cast*. The operator, `static_cast`, is a reserved keyword. It converts an expression from one type to another. The “static” in its name means the type is static, or fixed, at compile time.

You can convert any integer type to any other integer type. If the value is out of range for the target type, you get junk as a result. For example, the high-order bits may be discarded. Thus, you should always be careful when using `static_cast`. Be absolutely sure that you are not discarding important information.

If you cast a number to `bool`, the result is `false` if the number is zero or `true` if the number is not zero (just like the conversion that takes place when you use an integer as a condition).

Rewrite Listing 25-3 to overload print for short and signed char values too. Use type casting to force various values to different types and ensure that the results match your expectations. Take a look at Listing 25-4 to see one possible solution.

Listing 25-4. Using Type Casts

```
#include <iostream>
#include <locale>

typedef signed char byte;

void print(byte value)
{
  // The << operator treats signed char as a mutant char, and tries to
  // print a character. In order to print the value as an integer, you
  // must cast it to an integer type.
  std::cout << "byte=" << static_cast<int>(value) << '\n';
}
```

```
void print(short value)
{
  std::cout << "short=" << value << '\n';
}

void print(int value)
{
  std::cout << "int=" << value << '\n';
}

void print(long value)
{
  std::cout << "long=" << value << '\n';
}

void print(long long value)
{
  std::cout << "long long=" << value << '\n';
}

int main()
{
  std::cout.imbue(std::locale{""});
  print(0);
  print(0L);
  print(static_cast<short>(0));
  print(static_cast<byte>(0));
  print(static_cast<byte>(255));
  print(static_cast<short>(65535));
  print(32768);
  print(32768L);
  print(-32768);
  print(2147483647);
  print(-2147483647);
  print(2147483648);
  print(9223372036854775807);
  print(-9223372036854775807);
}
```

When I run Listing 25-4, I get `-1` for `static_cast<short>(65535)` and `static_cast<byte>(255)`. That's because the values are out of range for the target types. The resulting value is mere coincidence. In this case, it is related to the bit patterns that my particular compiler and platform happen to use. Different environments will yield different values.

Make Up Your Own Literals

Although C++ does not offer a built-in way to create a `short` literal, you can define your own literal suffix. Just as `42L` has type `long`, you can invent a suffix, say, `_S`, to mean `short`, so `42_S` is a compile-time constant of type `short`. Listing 25-5 shows how you can define your own literal suffix.

Listing 25-5. User-Defined Literal

```
#include <iostream>

short operator "" _S(unsigned long long value)
{
    return static_cast<short>(value);
}

void print(short s)
{
   std::cout << "short=" << s << '\n';
}

void print(int i)
{
   std::cout << "int=" << i << '\n';
}

int main()
{
   print(42);
   print(42_S);
}
```

When the user defines a literal, it is known as a *user-defined literal*, or UDL. The name of the literal must begin with an underscore. This will let the C++ standard define additional literals that don't start with an underscore without fear of interfering with the literals you define. You can define a UDL for integer, floating-point, and string types.

Integer Arithmetic

When you use `signed char` and `short` values or objects in an expression, the compiler always turns them into type `int`. It then performs the arithmetic or whatever operation you want to do. This is known as type *promotion*. The compiler *promotes* a `short` to an `int`. The result of arithmetic operations is also an `int`.

You can mix `int` and `long` in the same expressions. C++ converts the smaller type to match the larger type, and the larger type is the type of the result. This is known as type *conversion*, which is different from type promotion. (The distinction may seem arbitrary or trivial, but it's important. The next section will explain one of the reasons.) Remember: *Promote* `signed char` and `short` to `int`; *convert* `int` to `long`.

```
long big{2147483640};
short small{7};
std::cout << big + small; // promote small to type int; then convert it to long;
                          // the sum has type long
```

When you compare two integers, the same promotion and conversion occurs: the smaller argument is promoted or converted to the size of the larger argument. The result is always `bool`.

The compiler can convert any numeric value to `bool`; it considers this a conversion on the same level as any other integer conversion.

Overload Resolution

The two-step type conversion process may puzzle you. It matters when you have a set of overloaded functions, and the compiler has to decide which function to call. The first thing the compiler tries is to find an exact match. If it can't find one, it searches for a match after type promotion. Only if that fails does it search for a match allowing type conversion. Thus, it considers a match based only on type promotion to be better than type conversion. Listing 25-6 demonstrates the difference.

Listing 25-6. Overloading Prefers Type Promotion over Type Conversion

```
#include <iostream>

// print is overloaded for signed char, short, int and long
void print(signed char value)
{
  std::cout << "print(signed char = " << static_cast<int>(value) << ")\n";
}

void print(short value)
{
  std::cout << "print(short = " << value << ")\n";
}

void print(int value)
{
  std::cout << "print(int = " << value << ")\n";
}

void print(long value)
{
  std::cout << "print(long = " << value << ")\n";
}

// guess is overloaded for bool, int, and long
void guess(bool value)
{
  std::cout << "guess(bool = " << value << ")\n";
}

void guess(int value)
{
  std::cout << "guess(int = " << value << ")\n";
}

void guess(long value)
{
  std::cout << "guess(long = " << value << ")\n";
}
```

```
// error is overloaded for bool and long
void error(bool value)
{
  std::cout << "error(bool = " << value << ")\n";
}

void error(long value)
{
  std::cout << "error(long = " << value << ")\n";
}

int main()
{
  signed char byte{10};
  short shrt{20};
  int i{30};
  long lng{40};

  print(byte);
  print(shrt);
  print(i);
  print(lng);

  guess(byte);
  guess(shrt);
  guess(i);
  guess(lng);

  error(byte); // expected error
  error(shrt); // expected error
  error(i);    // expected error
  error(lng);
}
```

The first four lines of `main` call the `print` function. The compiler always finds an exact match and is happy. The next four lines call `guess`. When called with `signed char` and `short` arguments, the compiler promotes the arguments to `int` and finds an exact match with `guess(int i)`.

The last four lines call the aptly named function, `error`. The problem is that the compiler promotes `signed char` and `short` to `int` and then must convert `int` to either `long` or `bool`. It treats all conversions equally, thus it cannot decide which function to call, so it reports an error. Delete the three lines that I marked with "expected error," and the program works just fine, or add an overload for `error(int value)`, and everything will work.

The problem of ambiguous overload resolution is a difficult hurdle for new C++ programmers. It's also a difficult hurdle for many experienced C++ programmers. The exact rules for how C++ resolves overloaded names are complicated and subtle and will be covered in depth in Exploration 69. Avoid being clever about overloaded functions, and keep it simple. Most overloading situations are straightforward, but if you find yourself writing an overload for type `long`, be certain you also have an overload for type `int`.

Knowing about big integers helps with some programs, but others have to represent even larger numbers. The next Exploration examines how C++ works with floating point values.

EXPLORATION 26

Very Big and Very Little Numbers

Even the longest `long long` cannot represent truly large numbers, such as Avogadro's number ($6.02 \updownarrow 10^{23}$) or extremely small numbers, such as the mass of an electron ($9.1 \updownarrow 10^{-31}$ kg). Scientists and engineers use scientific notation, which consists of a mantissa (such as 6.02 or 9.1) and an exponent (such as 23 or –31), relative to a base (10).

Computers represent very large and very small numbers using a similar representation, known as *floating-point*. I know many of you have been waiting eagerly for this Exploration, as you've probably grown tired of using only integers, so let's jump in.

Floating-Point Numbers

Computers use floating-point numbers for very large and very small values. By sacrificing precision, you can gain a greatly extended range. However, never forget that the range and precision are limited. Floating-point numbers are not the same as mathematical real numbers, although they can often serve as useful approximations of real numbers.

Like its scientific notation counterpart, a floating-point number has a mantissa, also called a *significand*, a sign, and an exponent. The mantissa and exponent use a common *base* or *radix*. Although integers in C++ are always binary in their representation, floating-point numbers can use any base. Binary is a popular base, but some computers use 16 or even 10 as the base. The precise details are, as always, dependent upon the implementation. In other words, each C++ implementation uses its native floating-point format for maximum performance.

Floating-point values often come in multiple flavors. C++ offers single, double, and extended precision, called `float`, `double`, and `long double`, respectively. The difference is that `float` usually has less precision and a smaller range than `double`, and `double` usually has less precision and smaller range than `long double`. In exchange, `long double` usually requires more memory and computation time than `double`, which usually consumes more memory and computation time than `float`. On the other hand, an implementation is free to use the same representation for all three types.

Use `double`, unless there is some reason not to. Use `float` when memory is at a premium and you can afford to lose precision or `long double` when you absolutely need the extra precision or range and can afford to give up memory and performance.

A common binary representation of floating-point numbers is the IEC 60559 standard, which is better known as IEEE 754. Most likely, your desktop system has hardware that implements the IEC 60559 standard. For the sake of convenience, the following discussion describes only IEC 60559; however, never forget that C++ permits many floating-point representations. Mainframes and DSPs, for example, often use other representations.

An IEC 60559 `float` occupies 32 bits, of which 23 bits make up the mantissa and 8 bits form the exponent, leaving one bit for the mantissa's sign. The radix is 2, so the range of an IEC 60559 float is roughly 2^{-127} to 2^{127}, or 10^{-38} to 10^{38}. (I lied. Smaller numbers are possible, but the details are not germane to C++. If you are curious, look up *denormalization* in your favorite computer science reference.)

The IEC 60559 standard reserves some bit patterns for special values. In particular, if the exponent is all one bits, and the mantissa is all zero bits, the value is considered "infinity." It's not quite a mathematical infinity, but it does its best to pretend. Adding any finite value to infinity, for example, yields an answer of infinity. Positive infinity is always greater than any finite value, and negative infinity is always smaller than finite values.

If the exponent is all one bits, and the mantissa is not all zero bits, the value is considered as not-a-number, or *NaN*. NaN comes in two varieties: quiet and signaling. Arithmetic with quiet NaN always yields an NaN result. Using a signaling NaN results in a machine interrupt. How that interrupt manifests itself in your program is up to the implementation. In general, you should expect your program to terminate abruptly. Consult your compiler's documentation to learn the details. Certain arithmetic operations that have no meaningful result can also yield NaN, such as adding positive infinity to negative infinity.

Test whether a value is NaN by calling `std::isnan` (declared in `<cmath>`). Similar functions exist to test for infinity and other properties of floating-point numbers.

A `double` is similar in structure to a `float`, except it takes up 64 bits: 52 bits for the mantissa, 11 bits for the exponent, and 1 sign bit. A `double` can also have infinity and NaN values, with the same structural representation (that is, exponent all ones).

A `long double` is even longer than `double`. The IEC 60559 standard permits an extended double-precision format that requires at least 79 bits. Many desktop and workstation systems implement extended-precision, floating-point numbers using 80 bits (63 for the mantissa, 16 for the exponent, and 1 sign bit).

Floating-Point Literals

Any numeric literal with a decimal point or a decimal exponent represents a floating-point number. The decimal point is always `'.'`, regardless of locale. The exponent starts with the letter `e` or `E` and can be signed. No spaces are permitted in a numeric literal. For example:

```
3.1415926535897
31415926535897e-13
0.000314159265e4
```

By default, a floating-point literal has type `double`. To write a `float` literal, add the letter `f` or `F` after the number. For a `long double`, use the letter `l` or `L`, as in the following examples:

```
3.141592f
31415926535897E-13l
0.000314159265E+420L
```

As with `long int` literals, I prefer uppercase `L`, to avoid confusion with the digit `1`. Feel free to use `f` or `F`, but I recommend you pick one and stick with it. For uniformity with `L`, I prefer to use `F`.

If a floating-point literal exceeds the range of the type, the compiler will tell you. If you ask for a value at greater precision than the type supports, the compiler will silently give you as much precision as it can. Another possibility is that you request a value that the type cannot represent exactly. In that case, the compiler gives you the next higher or lower value.

For example, your program may have the literal `0.2F`, which seems like a perfectly fine real number, but as a binary floating-point value, it has no exact representation. Instead, it is approximately 0.0011001100_2. The difference between the decimal value and the internal value can give rise to unexpected results, the most common of which is when you expect two numbers to be equal and they are not. **Read Listing 26-1 and predict the outcome**.

Listing 26-1. Floating-Point Numbers Do Not Always Behave As You Expect

```
#include <cassert>
int main()
{
  float a{0.03F};
  float b{10.0F};
  float c{0.3F};
  assert(a * b == c);
}
```

What is your prediction?

____

What is the actual outcome?

____

Were you correct? ____________

The problem is that 0.03 and 0.3 do not have exact representations in binary, so if your floating-point format is binary (and most are), the values the computer uses are approximations of the real values. Multiplying 0.03 by 10 gives a result that is very close to 0.3, but the binary representation differs from that obtained by converting 0.3 to binary. (In IEC 60559 single-precision format, 0.03 * 10.0 gives $0.01110011001100110011001100_2$ and 0.3 is $0.01110011001100110011001101000_2$. The numbers are very close, but they differ in the 22nd significant bit.

Some programmers mistakenly believe that floating-point arithmetic is therefore "imprecise." On the contrary, floating-point arithmetic is exact. The problem lies only in the programmer's expectations, if you anticipate floating-point arithmetic to follow the rules of real-number arithmetic. If you realize that the compiler converts your decimal literals to other values, and computes with those other values, and if you understand the rules that the processor uses when it performs limited-precision arithmetic with those values, you can know exactly what the results will be. If this level of detail is critical for your application, you have to take the time to perform this level of analysis.

The rest of us, however, can continue to pretend that floating-point numbers and arithmetic are nearly real, without worrying overmuch about the differences. Just don't compare floating-point numbers for exact equality. (How to compare numbers for approximate equality is beyond the scope of this book. Visit the web site for links and references.)

Floating-Point Traits

You can query `numeric_limits` to reveal the size and limits of a floating-point type. You can also determine whether the type allows infinity or NaN. Listing 26-2 shows some code that displays information about a floating-point type.

Listing 26-2. Discovering the Attributes of a Floating-Point Type

```
#include <iostream>
#include <limits>
#include <locale>

int main()
{
  std::cout.imbue(std::locale{""});
  std::cout << std::boolalpha;
  // Change float to double or long double to learn about those types.
  typedef float T;
  std::cout << "min=" << std::numeric_limits<T>::min() << '\n'
       << "max=" << std::numeric_limits<T>::max() << '\n'
       << "IEC 60559? " << std::numeric_limits<T>::is_iec559 << '\n'
       << "max exponent=" << std::numeric_limits<T>::max_exponent << '\n'
       << "min exponent=" << std::numeric_limits<T>::min_exponent << '\n'
       << "mantissa places=" << std::numeric_limits<T>::digits << '\n'
       << "radix=" << std::numeric_limits<T>::radix << '\n'
       << "has infinity? " << std::numeric_limits<T>::has_infinity << '\n'
       << "has quiet NaN? " << std::numeric_limits<T>::has_quiet_NaN << '\n'
       << "has signaling NaN? " << std::numeric_limits<T>::has_signaling_NaN << '\n';

  if (std::numeric_limits<T>::has_infinity)
  {
    T zero{0};
    T one{1};
    T inf{std::numeric_limits<T>::infinity()};
    if (one/zero == inf)
      std::cout << "1.0/0.0 = infinity\n";
    if (inf + inf == inf)
      std::cout << "infinity + infinity = infinity\n";
  }
  if (std::numeric_limits<T>::has_quiet_NaN)
  {
    // There's no guarantee that your environment produces quiet NaNs for
    // these illegal arithmetic operations. It's possible that your compiler's
    // default is to produce signaling NaNs, or to terminate the program
    // in some other way.
    T zero{};
    T inf{std::numeric_limits<T>::infinity()};
    std::cout << "zero/zero = " << zero/zero << '\n';
    std::cout << "inf/inf = " << inf/inf << '\n';
  }
}
```

Modify the program so it prints information about **`double`**. Run it. Modify it again for **`long double`**, and run it. **Do the results match your expectations? ________________**

Floating-Point I/O

Reading and writing floating-point values depend on the locale. In the classic locale, the input format is the same as for an integer or floating-point literal. In a native locale, you must write the input according to the rules of the locale. In particular, the decimal separator must be that of the locale. Thousands-separators are optional, but if you use them, you must use the locale-specific character and correct placement.

Output is more complicated.

In addition to the field width and fill character, floating-point output also depends on the precision—the number of places after the decimal point—and the format, which can be fixed-point (without an exponent), scientific (with an exponent), or general (uses an exponent only when necessary). The default is general. Depending on the locale, the number may include separators for groups of thousands.

In the scientific and fixed formats (which you specify with a manipulator of the same name), the precision is the number of digits after the decimal point. In the general format, it is the maximum number of significant digits. Set the stream's precision with the `precision` member function or `setprecision` manipulator. The default precision is six. As usual, the manipulators that do not take arguments are declared in `<ios>`, so you get them for free with `<iostream>`, but `setprecision` requires that you include `<iomanip>`.

```
double const pi{3.141592653589792};
std::cout.precision(12);
std::cout << pi << '\n';
std::cout << std::setprecision(4) << pi << '\n';
```

In `scientific` format, the exponent is printed with a lowercase `'e'` (or `'E'`, if you use the `uppercase` manipulator), followed by the base 10 exponent. The exponent always has a sign (+ or -), and at least two digits, even if the exponent is zero. The mantissa is written with one digit before the decimal point. The precision determines the number of places after the decimal point.

In `fixed` format, no exponent is printed. The number is printed with as many digits before the decimal point as needed. The precision determines the number of places after the decimal point. The decimal point is always printed.

The default format is the general format, which means printing numbers nicely without sacrificing information. If the exponent is less than or equal to –4, or if it is greater than the precision, the number is printed in scientific format. Otherwise, it is printed without an exponent. However, unlike conventional fixed-point output, trailing zeros are removed after the decimal point. If after removal of the trailing zeros the decimal point becomes the last character, it is also removed.

When necessary, values are rounded off to fit within the allotted precision.

A new format in C++ 11 is `hexfloat`. The value is printed in hexadecimal, which lets you discover the exact value on systems with binary or base 16 representations. Because the letter `'e'` is a valid hexadecimal value, the exponent is marked with the letters `'p'` or `'P'`.

The easiest way to specify a particular output format is with a manipulator: `scientific`, `fixed`, or `hexfloat`. Like the precision, the format persists in the stream's state until you change it. (Only width resets after an output operation.) Unfortunately, once you set the format, there is no easy way to revert to the default general format. To do that, you must use a member function, and a clumsy one at that, as shown in the following:

```
std::cout << std::scientific << large_number << '\n';
std::cout << std::fixed << small_number << '\n';
std::cout.unsetf(std::ios_base::floatfield);
std::cout << number_in_general_format << '\n';
```

Complete Table 26-1, showing exactly how each value would be printed in each format, in the classic locale. I filled in the first row for your convenience.

Table 26-1. *Floating-Point Output*

Value	Precision	Scientific	Fixed	Hexfloat	General
123456.789	6	`1.234568e5`	`123456.789000`	`0x1.e240cap+16`	`123457`
1.23456789	4	____________	____________	____________	____________
123456789	2	____________	____________	____________	____________
–1234.5678e9	5	____________	____________	____________	____________

After you have filled in the table with your predictions, **write a program that will test your predictions**, then run it and see how well you did. Compare your program with Listing 26-3.

Listing 26-3. Demonstrating Floating-Point Output

```
#include <iostream>

/// Print a floating-point number in three different formats.
/// @param precision the precision to use when printing @p value
/// @param value the floating-point number to print
void print(int precision, float value)
{
  std::cout.precision(precision);
  std::cout << std::scientific << value << '\t'
            << std::fixed      << value << '\t'
            << std::hexfloat   << value << '\t';

  // Set the format to general.
  std::cout.unsetf(std::ios_base::floatfield);
  std::cout << value << '\n';
}

/// Main program.
int main()
{
  print(6, 123456.789f);
  print(4, 1.23456789f);
  print(2, 123456789.f);
  print(5, -1234.5678e9f);
}
```

The precise values can differ from one system to another, depending on the floating-point representation. For example, `float` on most systems cannot support the full precision of nine decimal digits, so you should expect some fuzziness in the least significant digits of the printed result. In other words, unless you want to sit down and do some serious binary computation, you cannot easily predict exactly what the output will be in every case. Table 26-2 shows the output from Listing 26-3, when run on a typical IEC 60559–compliant system.

Table 26-2. *Results of Printing Floating-Point Numbers*

Value	Precision	Scientific	Fixed	Hexfloat	General
123456.789	6	1.234568e+05	123456.789062	0x1.e240cap+16	123457
1.23456789	4	1.2346e+00	1.2346	0x1.3c0ca4p+0	1.235
123456789	2	1.23e+08	123456792.00	0x1.d6f346p+26	1.2e+08
–1234.5678e9	5	-1.23457e+12	-1234567823360.00000	-0x1.1f71fap+40	-1.2346e+12

Some applications are never required to use floating-point numbers; others need them a lot. Scientists and engineers, for example, depend on floating-point arithmetic and math functions and must understand the subtleties of working with these numbers. C++ has everything you need for computation-intensive programming. Although the details are beyond the scope of this book, interested readers should consult a reference for the `<cmath>` header and the transcendental and other functions that it provides. The `<cfenv>` header contains functions and related declarations to let you adjust the rounding mode and other aspects of the floating-point environment. If you cannot find information about `<cfenv>` in a C++ reference, consult a C 99 reference for the `<fenv.h>` header.

The next Exploration takes a side trip to a completely different topic, explaining the strange comments—the extra slashes (`///`) and stars (`/**`)—that I've used in so many programs.

EXPLORATION 27

Documentation

This Exploration is a little different from the others. Instead of covering part of the C++ standard, it examines a third-party tool called Doxygen. Feel free to skip this Exploration, but understand that this is where I explain the strange comments you sometimes see in the code listings.

Doxygen

Doxygen is a free (in cost and license) tool that reads your source code, looks for comments that follow a certain structure, and extracts information from the comments and from the code to produce documentation. It produces output in a number of formats: HTML, RTF (rich text format), LaTeX, UNIX man pages, and XML.

Java programmers may be familiar with a similar tool called javadoc. The javadoc tool is standard in the Java Software Development Kit, whereas Doxygen has no relationship with the C++ standard or with any C++ vendor. C++ lacks a standard for structured documentation, so you are free to do anything you want. For example, Microsoft defines its own conventions for XML tags in comments, which is fine, if you plan to work entirely within the Microsoft .NET environment. For other programmers, I recommend using tools that have more widespread and portable use. The most popular solution is Doxygen, and I think every C++ programmer should know about it, even if you decide not to use it. That's why I include this Exploration in the book.

Structured Comments

Doxygen heeds comments that follow a specific format.

- One-line comments start with an extra slash or exclamation mark: `///` or `//!`
- Multiline comments start with an extra asterisk or exclamation mark: `/**` or `/*!`

Also, Doxygen recognizes some widespread comment conventions and decorations. For example, it ignores a line of slashes.

```
/////////////////////////////////////////////////////////////////////////////
```

A multiline comment can begin with a row full of asterisks.

```
/*****************************************************************************
```

And a line in a multiline comment can begin with an asterisk.

```
/****************************************************************************
 * This is a structured comment for Doxygen.                                *
 ****************************************************************************/
```

Within a structured comment is where you document the various entities in your program: functions, types, variables, and so on.

The convention is that the comment immediately before a declaration or definition applies to the entity being declared or defined. Sometimes, you want to put the comment after the declaration, such as a one-line description of a variable. To do that, use a "less-than" (`<`) sign at the start of the comment.

```
double const c = 299792458.0;            ///< speed of light in m/sec
```

Documentation Tags and Markdown

Doxygen has its own markup language that utilizes *tags*. A tag can start with a backslash character (`\return`) or an "at sign" (`@return`). Some tags take arguments and some don't. In addition to its own tags, you can also use a subset of HTML or Markdown (a wiki-like syntax that is easy to read and write). The most useful tags, markup, and Markdown are the following:

@b word

Mark up *word* in boldface. You can also use HTML markup, `<b>`*phrase*`</b>`, which is helpful when *phrase* contains spaces, or use Markdown, by enclosing the text in asterisks: `*phrase*`.

`@brief` one-sentence-description

Describe an entity briefly. Entities have brief and detailed documentation. Depending on how you configure Doxygen, the brief documentation can be the first sentence of the entity's full documentation, or you can require an explicit `@brief` tag. In either case, the rest of the comment is the detailed documentation for the entity.

`@c` word

Treat *word* as a code fragment and set it in a fixed-pitch typeface. You can also use HTML markup, `<tt>`*phrase*`</tt>`, or use backticks for Markdown, `` `phrase` ``.

`@em` word

Emphasize *word* in italics. You can also use HTML tags, `<em>`*phrase*`</em>`, or underscores for Markdown: *_phrase_*.

`@file` file name

Presents an overview of the source file. The detailed description can describe the purpose of the file, revision history, and other global documentation. The *file name* is optional; without it, Doxygen uses the file's real name.

`@link` entity text `@endlink`

Create a hyperlink to the named *entity*, such as a file. I use `@link` on my `@mainpage` to create a table of contents to the most important files in the project or to the sole file. Markdown offers a variety of ways to create links, such as `[text](entity)`.

@mainpage title

Present an overview of the entire project for the index or cover page. You can put `@mainpage` in any source file, or even set aside a separate file just for the comment. In small projects, I place `@mainpage` in the same source file as the `main` function, but in large projects, I use a separate file.

@p name

Set *name* in a fixed-pitch typeface to distinguish it as a function parameter.

@par title

Start a new paragraph. If you supply a one-line *title*, it becomes the paragraph heading. A blank line also separates paragraphs.

@param name description

Document a function parameter named *name*. If you want to refer to this parameter elsewhere in the function's documentation, use `@p` *name*.

@post postcondition

Document a postcondition for a function. A postcondition is a Boolean expression that you can assume to be true when the function returns (assuming all preconditions were true). C++ lacks any formal mechanism for enforcing postconditions (other than `assert`), so documenting postconditions is crucial, especially for library writers.

@pre precondition

Document a precondition for a function. A precondition is a Boolean expression that must be true before the function is called, or else the function is not guaranteed to work properly. C++ lacks any formal mechanism for enforcing preconditions (other than `assert`), so documenting preconditions is crucial, especially for library writers.

@return description

Document what a function returns.

@see xref

Insert a cross-reference to an entity named *xref*. Doxygen looks for references to other documented entities within the structured comment. When it finds one, it inserts a hyperlink (or text cross-reference, depending on the output format). Sometimes, however, you have to insert an explicit reference to an entity that is not named in the documentation, so you can use `@see`.

You can suppress automatic hyperlink creation by prefacing a name with `%`.

@&, @@, @\, @%, @<

Escapes a literal character (&, @, \, %, or <), to prevent interpretation by Doxygen or HTML.

Doxygen is very flexible, and you have many, many ways to format your comments using native Doxygen tags, HTML, or Markdown. This book's web site has links to the main Doxygen page, where you can find more information about the tool and download the software. Most Linux users already have Doxygen; other users can download Doxygen for their favorite platform.

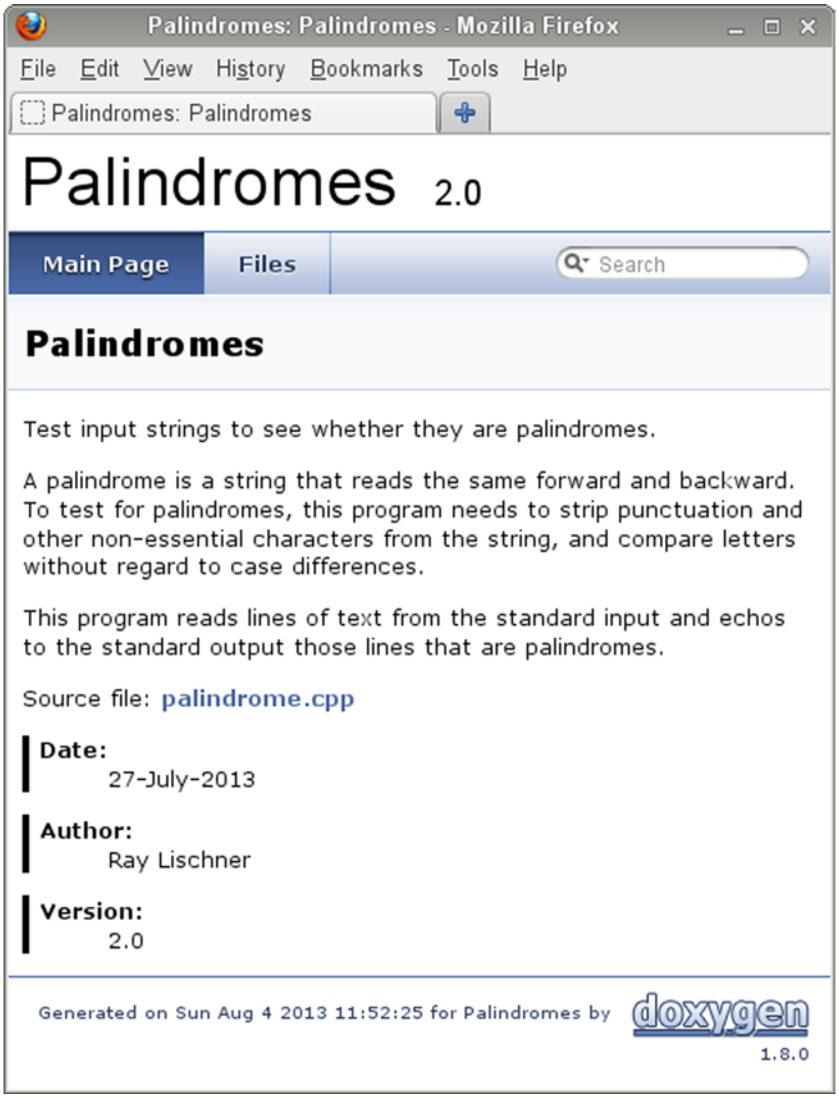
Listing 27-1 shows a few of the many ways you can use Doxygen.

Listing 27-1. Documenting Your Code with Doxygen

```
/** @file
 * @brief Test strings to see whether they are palindromes.
 * Read lines from the input, strip non-letters, and check whether
 * the result is a palindrome. Ignore case differences when checking.
 * Echo palindromes to the standard output.
 */

/** @mainpage Palindromes
 * Test input strings to see whether they are palindromes.
 *
 * A _palindrome_ is a string that reads the same forward and backward.
 * To test for palindromes, this program needs to strip punctuation and
 * other non-essential characters from the string, and compare letters without
 * regard to case differences.
 *
 * This program reads lines of text from the standard input and echoes
 * to the standard output those lines that are palindromes.
 *
 * Source file: [palindrome.cpp](palindrome.cpp)

 *
 * @date 27-July-2013
 * @author Ray Lischner
 * @version 2.0
 */
#include <algorithm>
#include <iostream>
#include <iterator>
#include <locale>
#include <string>

/** @brief Test for non-letter.
 * Test the character @p ch in the global locale.
 * @param ch the character to test
 * @return true if @p ch is not a letter
 */
bool non_letter(char ch)
{
  return not std::isalnum(ch, std::locale{});
}
```

```
/** @brief Convert to lowercase.
 * Use a canonical form by converting to uppercase first,
 * and then to lowercase. This approach does not solve the eszet
 * problem (German eszet is a lowercase character that converts
 * to two uppercase characters), but it's the best we can do in
 * standard C++.
 *
 * All conversions use the global locale.
 *
 * @param ch the character to test
 * @return the character converted to lowercase
 */
char lowercase(char ch)
{
  return std::tolower(ch, std::locale{});
}

/** @brief Compare two characters without regard to case.
 * The comparison is limited by the `lowercase()` function.
 * @param a one character to compare
 * @param b the other character to compare
 * @return `true` if the characters are the same in lowercase,
 *         `false` if they are different.
 */
bool is_same_char(char a, char b)
{
  return lowercase(a) == lowercase(b);
}

/** @brief Determine whether @p str is a palindrome.
 * Only letter characters are tested. Spaces and punctuation don't count.
 * Empty strings are not palindromes because that's just too easy.
 * @param str the string to test
 * @return `true` if @p str is the same forward and backward and
 *     `not str.empty()`

 */
bool is_palindrome(std::string str)
{
  std::string::iterator end{std::remove_if(str.begin(), str.end(), non_letter)};
  std::string rev{str.begin(), end};
  std::reverse(rev.begin(), rev.end());
  return not rev.empty() and std::equal(str.begin(), end, rev.begin(), is_same_char);
}

/** @brief Main program.
 * Set the global locale to the user's native locale. Then imbue the I/O streams
 * with the native locale.
 */
int main()
{
  std::locale::global(std::locale{""});
  std::cin.imbue(std::locale{});
  std::cout.imbue(std::locale{});

  std::string line{};
  while (std::getline(std::cin, line))
    if (is_palindrome(line))
      std::cout << line << '\n';
}
```

Figure 27-1 shows the main page as it appears in a web browser.


Palindromes: Palindromes - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Palindromes: Palindromes
Main Page
Files
Search

Palindromes 2.0

Palindromes

Test input strings to see whether they are palindromes.

A palindrome is a string that reads the same forward and backward. To test for palindromes, this program needs to strip punctuation and other non-essential characters from the string, and compare letters without regard to case differences.

This program reads lines of text from the standard input and echos to the standard output those lines that are palindromes.

Source file: palindrome.cpp

Date:
27-July-2013

Author:
Ray Lischner

Version:
2.0

Generated on Sun Aug 4 2013 11:52:25 for Palindromes by doxygen 1.8.0

Figure 27-1. *Main page of the palindrome documentation*

Using Doxygen

Instead of taking lots of command-line arguments, Doxygen uses a configuration file, typically named `Doxyfile`, in which you can put all that juicy information. Among the information in the configuration file are the name of the project, which files to examine for comments, which output format or formats to generate, and a variety of options you can use to tweak and adjust the output.

Because of the plethora of options, Doxygen comes with a wizard, `doxywizard`, to help generate a suitable configuration file, or you can just run the command line `doxygen` utility with the `-g` switch, to generate a default configuration file that has lots of comments to help you understand how to customize it.

Once you have configured Doxygen, running the program is trivial. Simply run `doxygen`, and away it goes. Doxygen does a reasonable job at parsing C++, which is a complicated and difficult language to parse, but it sometimes gets confused. Pay attention to the error messages, to see if it had any difficulties with your source files.

The configuration file dictates the location of the output. Typically, each output format resides in its own subdirectory. For example, the default configuration file stores HTML output in the `html` directory. Open the `html/index.html` file in your favorite browser, to check out the results.

Download and install Doxygen on your system.

Add Doxygen comments to one of your programs. Configure and run Doxygen.

Future programs will continue to use Doxygen comments sporadically, when I think the comments help you understand what the program does. In general, however, I try to avoid them in the book, because the text usually explains things well enough, and I don't want to waste any space. The programs that accompany the book, however, have more complete Doxygen comments.

EXPLORATION 28

Project 1: Body-Mass Index

It's project time! Body-mass index (BMI) is a measurement some health-care professionals use to determine whether a person is overweight, and if so, by how much. To compute BMI, you need a person's weight in kilograms and height in meters. The BMI is simply weight/height^2, converted to a unitless value.

Your task is to write a program that reads records, prints the records, and computes some statistics. The program should start by asking for a threshold BMI. Only records with a BMI greater than or equal to the threshold will be printed. Each record needs to consist of a person's name (which may contain spaces), weight in kilograms, height in centimeters (not meters), and the person's sex (`'M'` or `'F'`). Let the user enter the sex in uppercase or lowercase. Ask the user to enter the height in centimeters, so you can compute the BMI using integers. You will have to adjust the formula to allow for centimeters instead of meters.

Print each person's BMI immediately after reading his or her record. After collecting information for everyone, print two tables—one for men, one for women—based on the data. Use an asterisk after the BMI rating to mark records for which the number meets or exceeds the threshold. After each table, print the mean (average) and median BMI. (Median is the value at which half the BMI values are less than the median and half are greater than the median. If the user enters an even number of records, take the mean of the two values in the middle.) Compute individual BMI values as integers. Compute the mean and median BMI values as floating-point numbers, and print the mean with one place after the decimal point.

Listing 28-1 shows a sample user session. User input is in **boldface**.

Listing 28-1. Sample User Session with the BMI Program

```
$ bmi
Enter threshold BMI: 25
Enter name, height (in cm), and weight (in kg) for each person:
Name 1: Ray Lischner
Height (cm): 180
Weight (kg): 90
Sex (m or f): m
BMI = 27
Name 2: A. Nony Mouse
Height (cm): 120
Weight (kg): 42
Sex (m or f): F
BMI = 29
Name 3: Mick E. Mouse
Height (cm): 30
Weight (kg): 2
Sex (m or f): M
BMI = 22
Name 4: A. Nony Lischner
```

```
Height (cm): 150
Weight (kg): 55
Sex (m or f): m
BMI = 24
Name 5: No One
Height (cm): 250
Weight (kg): 130
Sex (m or f): f
BMI = 20
Name 6: ^Z

Male data
Ht(cm) Wt(kg) Sex  BMI  Name
   180     90  M    27* Ray Lischner
    30      2  M    22  Mick E. Mouse
   150     55  M    24  A. Nony Lischner
Mean BMI = 24.3
Median BMI = 24

Female data
Ht(cm) Wt(kg) Sex  BMI  Name
   120     42  F    29* A. Nony Mouse
   250    130  F    20  No One
Mean BMI = 24.5
Median BMI = 24.5
```

Hints

Here are some hints, in case you need them.

- Keep track of the data in separate vectors, e.g., `heights`, `weights`, `sexes`, `names`, `bmis`.
- Use the native locale for all input and output.
- Divide the program into functions, e.g., `compute_bmi` to compute the BMI from weight and height.
- You can write this program using nothing more than the techniques that we have covered so far, but if you know other techniques, feel free to use them. The next set of Explorations will present language features that will greatly facilitate writing this kind of program.
- The complete source code for my solution is available with the other files that accompany this book, but don't peek until after you have written the program yourself.

EXPLORATION 29

Custom Types

One of the key design goals for C++ was that you should be able to define brand-new types that look and act similar to the built-in types. Do you need tri-state logic? Write your own `tribool` type. Need arbitrary-precision arithmetic? Write your own `bigint` type. This Exploration introduces some of the language features that let you define custom types. Subsequent Explorations delve deeper into these topics.

Defining a New Type

Let's consider a scenario in which you want to define a type, `rational`, to represent rational numbers (fractions). A rational number has a numerator and a denominator, both integers. Ideally, you would be able to add, subtract, multiply, and divide rational numbers in the same fashion you can with the built-in numeric types. You should also be able to mix rational numbers and other numeric types in the same expression. (Our `rational` type is different from the `std::ratio` type, which represents a compile-time constant; our `rational` type can change value at runtime.)

The I/O streams should be able to read and write rational numbers in some reasonable manner. The input operator should accept as valid input anything the output operator produces. The I/O operators should heed the stream's flags and related settings, such as field width and fill character, so you can format neatly aligned columns of rational numbers the same way you did for integers in Exploration 8.

You should be able to assign any numeric value to a `rational` variable and convert a `rational` value to any built-in numeric type. Naturally, converting a rational number to an integer variable would result in truncation to an integer. One can argue that conversion should be automatic, similar to conversion from floating-point to integer. A counter argument is that automatic conversions that discard information were a mistake in the original C language design, and one not to be duplicated. Instead, conversions that discard information should be made explicit and clear. I prefer the latter approach.

This is a lot to tackle at once, so let's begin slowly.

The first step is to decide how to store a rational number. You have to store a numerator and a denominator, both as integers. What about negative numbers? Choose a convention, such as the numerator gets the sign of the entire value, and the denominator is always positive. Listing 29-1 shows a basic `rational` type definition.

Listing 29-1. Defining a Custom rational Type

```
/// Represent a rational number.
struct rational
{
  int numerator;     ///< numerator gets the sign of the rational value
  int denominator;   ///< denominator is always positive
};
```

The definition starts with the struct keyword. C programmers recognize this as a structure definition—but hang on, there's much more to follow.

The contents of the rational type look like definitions for variables named numerator and denominator, but they work a little differently. Remember that Listing 29-1 shows a type definition. In other words, the compiler remembers that rational names a type, but it does not allocate any memory for an object, for numerator, or for denominator. In C++ parlance, numerator and denominator are called *data members*; some other languages call them instance variables or fields.

Notice the semicolon that follows the closing curly brace. Type definitions are different from compound statements. If you forget the semicolon, the compiler will remind you, sometimes quite rudely, while referring to a line number several lines after the one where the semicolon belongs.

When you define an object with type rational, the object stores the numerator and denominator members. Use the dot (.) operator to access the members (which you have been doing throughout this book), as shown in Listing 29-2.

Listing 29-2. Using a Custom Type and Its Members

```
#include <iostream>

/// Represent a rational number.
struct rational
{
  int numerator;     ///< numerator gets the sign of the rational value
  int denominator;   ///< denominator is always positive
};

int main()
{
  rational pi{};
  pi.numerator = 355;
  pi.denominator = 113;
  std::cout << "pi is approximately " << pi.numerator << "/" << pi.denominator << '\n';
}
```

That's not terribly exciting, is it? The rational type just sits there, lifeless. You know that many types in the standard library have member functions, such as std::ostream's width member function, which allows you to write std::cout.width(4). The next section shows how to write your own member functions.

Member Functions

Let's add a member function to rational that reduces the numerator and denominator by their greatest common divisor. Listing 29-3 shows the sample program, with the reduce() member function.

Listing 29-3. Adding the reduce Member Function

```
#include <cassert>
#include <cstdlib>
#include <iostream>

/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
int gcd(int n, int m)
{
  n = std::abs(n);
  while (m != 0) {
    int tmp{n % m};
    n = m;
    m = tmp;
  }
  return n;
}

/// Represent a rational number.
struct rational
{
  /// Reduce the numerator and denominator by their GCD.
  void reduce()
  {
    assert(denominator != 0);
    int div{gcd(numerator, denominator)};
    numerator = numerator / div;
    denominator = denominator / div;
  }

  int numerator;     ///< numerator gets the sign of the rational value
  int denominator;   ///< denominator is always positive
};

int main()
{
  rational pi{};
  pi.numerator = 1420;
  pi.denominator = 452;
  pi.reduce();
  std::cout << "pi is approximately " << pi.numerator << "/" << pi.denominator << '\n';
}
```

Notice how the `reduce()` member function looks just like an ordinary function, except its definition appears within the `rational` type definition. Also notice how `reduce()` can refer to the data members of `rational`. When you call the `reduce()` member function, you must supply an object as the left-hand operand of the dot (`.`) operator (such as `pi` in Listing 29-3). When the `reduce()` function refers to a data member, the data member is taken from that left-hand operand. Thus, `numerator = numerator / div` has the effect of `pi.numerator = pi.numerator/div`.

The `gcd` function is a free function. You can call it with any two integers, unrelated to `rational` numbers. I could have defined `gcd` as a member function, but nothing about the function ties it to rational numbers. It does not access any members of `rational`. By making it a free function, you can reuse it throughout your program, anywhere you have to compute the greatest common divisor of two integers. If you aren't sure whether a function should be free or a member, err on the side of making it free.

A member function can also call other member functions that are defined in the same type. Try it yourself: **add the assign() member function**, which takes a numerator and denominator as two parameters and assigns them to their respective data members and calls reduce(). This spares the user of rational the additional step (and possible error of neglecting the call to reduce()). Let the return type be void. **Write your member function below:**

____

____

____

____

____

____

Listing 29-4 presents the entire program, with my assign member function in boldface.

Listing 29-4. Adding the assign Member Function

```
#include <cassert>
#include <cstdlib>
#include <iostream>

/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
int gcd(int n, int m)
{
  n = std::abs(n);
  while (m != 0) {
    int tmp(n % m);
    n = m;
    m = tmp;
  }
  return n;
}

/// Represent a rational number.
struct rational
{
  /// Assign a numerator and a denominator, then reduce to normal form.
  /// @param num numerator
  /// @param den denominator
  /// @pre denominator > 0
  void assign(int num, int den)
  {
    numerator = num;
    denominator = den;
    reduce();
  }

  /// Reduce the numerator and denominator by their GCD.
  void reduce()
  {
    assert(denominator != 0);
    int div{gcd(numerator, denominator)};
    numerator = numerator / div;
    denominator = denominator / div;
  }

  int numerator;     ///< numerator gets the sign of the rational value
  int denominator;   ///< denominator is always positive
};

int main()
{
  rational pi{};
  pi.assign(1420, 452);
  std::cout << "pi is approximately " << pi.numerator << "/" << pi.denominator << '\n';
}
```

Notice how simple the `main` program is now. Hiding details, such as `reduce()`, helps keep the code clean, readable, and maintainable.

Notice one other subtle detail: the definition of `assign()` precedes `reduce()`, even though it calls `reduce()`. We need one minor adjustment to the rule that the compiler must see at least a declaration of a name before you can use that name. Members of a new type can refer to other members, regardless of the order of declaration within the type. In all other situations, you must supply a declaration prior to use.

Being able to assign a numerator and denominator in one step is a fine addition to the `rational` type, but even more important is being able to initialize a `rational` object. Recall from Exploration 5 my admonition to ensure that all objects are properly initialized. The next section demonstrates how to add support for initializers to `rational`.

Constructors

Wouldn't it be nice to be able to initialize a `rational` object with a numerator and denominator, and have them properly reduced automatically? You can do that by writing a special member function that looks and acts a little like `assign`, except the name is the same as the name of the type (`rational`), and the function has no return type or return value. Listing 29-5 shows how to write this special member function.

Listing 29-5. Adding the Ability to Initialize a `rational` Object

```
#include <cassert>
#include <cstdlib>
#include <iostream>

/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
int gcd(int n, int m)
{
  n = std::abs(n);
  while (m != 0) {
    int tmp(n % m);
    n = m;
    m = tmp;
  }
  return n;
}
```

```
/// Represent a rational number.
struct rational
{
  /// Construct a rational object, given a numerator and a denominator.
  /// Always reduce to normal form.
  /// @param num numerator
  /// @param den denominator
  /// @pre denominator > 0
  rational(int num, int den)
  : numerator{num}, denominator{den}
  {
    reduce();
  }

  /// Assign a numerator and a denominator, then reduce to normal form.
  /// @param num numerator
  /// @param den denominator
  /// @pre denominator > 0
  void assign(int num, int den)
  {
    numerator = num;
    denominator = den;
    reduce();
  }

  /// Reduce the numerator and denominator by their GCD.
  void reduce()
  {
    assert(denominator != 0);
    int div{gcd(numerator, denominator)};
    numerator = numerator / div;
    denominator = denominator / div;
  }

  int numerator;     ///< numerator gets the sign of the rational value
  int denominator;   ///< denominator is always positive
};

int main()
{
  rational pi{1420, 452};
  std::cout << "pi is about " << pi.numerator << "/" << pi.denominator << '\n';
}
```

Notice the definition of the pi object. The variable takes arguments in its initializer, and those two arguments are passed to the special initializer function in the same manner as function arguments. This special member function is called a *constructor*.

A constructor looks very much like a normal function, except that it doesn't have a return type. Also, you can't choose a name but must use the type name. And then there's that extra line that starts with a colon. This extra bit of code initializes the data members in the same manner as initializing a variable. After all the data members are initialized, the body of the constructor runs in the same manner as any member function body.

The initializer list is optional. Without it, data members are left uninitialized—this is a bad thing, so don't do it.

Modify the `rational` type so it accepts a negative denominator. If the denominator is negative, change it to positive and also change the sign of the numerator. Thus, `rational{-710, -227}` would have the same value as `rational{710, 227}`.

You can choose to perform the modification in any one of a number of places. Good software design practice dictates that the change should occur in exactly one spot, and all other functions should call that one. Therefore, I suggest modifying `reduce()`, as shown in Listing 29-6.

Listing 29-6. Modifying the `reduce` Member Function to Accept a Negative Denominator

```
/// Reduce the numerator and denominator by their GCD.
void reduce()
{
  assert(denominator != 0);
  if (denominator < 0)
  {
    denominator = -denominator;
    numerator = -numerator;
  }
  int div{gcd(numerator, denominator)};
  numerator = numerator / div;
  denominator = denominator / div;
}
```

Overloading Constructors

You can overload a constructor the same way you overload an ordinary function. All the overloaded constructors have the same name (that is, the name of the type), and they must differ in the number or type of the parameters. For example, you can add a constructor that takes a single integer argument, implicitly using 1 as the denominator.

You have a choice of two ways to write an overloaded constructor. You can have one constructor call another one, or you can have the constructor initialize all the members, just like the first constructor you wrote. To call another constructor, use the type name in the initializer list.

```
rational(int num) : rational{num, 1} {}
```

Or initialize each member directly.

```
rational(int num)
: numerator{num}, denominator{1}
{}
```

When you want multiple constructors to share a common behavior, delegating the work to a common constructor makes a lot of sense. For example, you can have a single constructor call `reduce()`, and every other constructor can call that one constructor, thereby ensuring that no matter how you construct the `rational` object, you know it has been reduced.

On the other hand, when the denominator is 1, there is no need to call `reduce()`, so you might prefer the second form, initializing the data members directly. The choice is yours.

A constructor that calls another constructor is known as a *delegating constructor* because it delegates its work to the other constructor.

I'm sure you can see many deficiencies in the current state of the `rational` type. It has several that you probably missed too. Hang on; the next Exploration starts improving the type. For example, you may want to test your modification by comparing two `rational` objects, to see if they are equal. To do so, however, you have to write a custom == operator, which is the subject of the next Exploration.

Overloading Operators

This Exploration continues the study of writing custom types. An important aspect of making a custom type behave seamlessly with built-in types is ensuring that the custom types support all the expected operators—arithmetic types must support arithmetic operators, readable and writable types must support I/O operators, and so on. Fortunately, C++ lets you overload operators in much the same manner as overloading functions.

Comparing Rational Numbers

In the previous Exploration, you began to write a `rational` type. After making a modification to it, an important step is testing the modified type, and an important aspect of internal testing is the equality (`==`) operator. C++ lets you define a custom implementation for almost every operator, provided at least one operand has a custom type. In other words, you can't redefine integer division to yield a `rational` result, but you can define division of an integer by a `rational` number, and vice versa.

To implement a custom operator, write a normal function, but for the function name, use the `operator` keyword, followed by the operator symbol, as shown in the code excerpt in Listing 30-1.

Listing 30-1. Overloading the Equality Operator

```
/// Represent a rational number.
struct rational
{
  /// Construct a rational object, given a numerator and a denominator.
  /// Always reduce to normal form.
  /// @param num numerator
  /// @param den denominator
  /// @pre denominator > 0
  rational(int num, int den)
  : numerator{num}, denominator{den}
  {
    reduce();
  }

  /// Assign a numerator and a denominator, then reduce to normal form.
  /// @param num numerator
  /// @param den denominator
  /// @pre denominator > 0
  void assign(int num, int den)
  {
    numerator = num;
    denominator = den;
    reduce();
  }

  /// Reduce the numerator and denominator by their GCD.
  void reduce()
  {
    assert(denominator != 0);
    if (denominator < 0)
    {
      denominator = -denominator;
      numerator = -numerator;
    }
    int div{gcd(numerator, denominator)};
    numerator = numerator / div;
    denominator = denominator / div;
  }

  int numerator;     ///< numerator gets the sign of the rational value
  int denominator;   ///< denominator is always positive
};

/// Compare two rational numbers for equality.
/// @pre @p a and @p b are reduced to normal form
bool operator==(rational const& a, rational const& b)
{
  return a.numerator == b.numerator and a.denominator == b.denominator;
}

/// Compare two rational numbers for inequality.
/// @pre @p a and @p b are reduced to normal form
bool operator!=(rational const& a, rational const& b)
{
  return not (a == b);
}
```

One of the benefits of reducing all `rational` numbers is that it makes comparison easier. Instead of checking whether `3/3` is the same as `6/6`, the constructor reduces both numbers to `1/1`, so it is just a matter of comparing the numerators and denominators. Another trick is defining `!=` in terms of `==`. There's no point in your making extra work for yourself, so confine the actual logic of comparing `rational` objects to one function and call it from another function. If you worry about the performance overhead of calling an extra layer of functions, use the `inline` keyword, as shown in Listing 30-2.

Listing 30-2. Using `inline` for Trivial Functions

```
/// Compare two rational numbers for equality.
/// @pre @p a and @p b are reduced to normal form
bool operator==(rational const& a, rational const& b)
{
  return a.numerator == b.numerator and a.denominator == b.denominator;
}
```

```
/// Compare two rational numbers for inequality.
/// @pre @p a and @p b are reduced to normal form
inline bool operator!=(rational const& a, rational const& b)
{
  return not (a == b);
}
```

The `inline` keyword is a hint to the compiler that you would like the function expanded at the point of call. If the compiler decides to heed your wish, the resulting program will not have any identifiable function named `operator!=` in it. Instead, every place where you use the `!=` operator with `rational` objects, the function body is expanded there, resulting in a call to `operator==`.

To implement the `<` operator, you need a common denominator. Once you implement `operator<`, you can implement all other relational operators in terms of `<`. You can choose any of the relational operators (`<`, `>`, `<=`, `>=`) as the fundamental operator and implement the others in terms of the fundamental. The convention is to start with `<`. Listing 30-3 demonstrates `<` and `<=`.

Listing 30-3. Implementing the `<` Operator for `rational`

```
/// Compare two rational numbers for less-than.
bool operator<(rational const& a, rational const& b)
{
  return a.numerator * b.denominator < b.numerator * a.denominator;
}

/// Compare two rational numbers for less-than-or-equal.
inline bool operator<=(rational const& a, rational const& b)
{
  return not (b < a);
}
```

Implement `>` and `>=` in terms of `<`.

Compare your operators with Listing 30-4.

Listing 30-4. Implementing the `>` and `>=` Operators in Terms of `<`

```
/// Compare two rational numbers for greater-than.
inline bool operator>(rational const& a, rational const& b)
{
  return b < a;
}

/// Compare two rational numbers for greater-than-or-equal.
inline bool operator>=(rational const& a, rational const& b)
{
  return not (b > a);
}
```

Then write a test program. To help you write your tests, download the `test.hpp` file and add `#include "test.hpp"` to your program. Call the `TEST()` function as many times as you need, passing a Boolean expression as the sole argument. If the argument is true, the test passed. If the argument is false, the test failed, and the `TEST` function prints a suitable message. Thus, you may write tests, such as the following:TEST(rational{2, 2} == rational{5, 5});TEST(rational{6,3} > rational{10, 6});

The all-capital name, TEST, tells you that TEST is different from an ordinary function. In particular, if the test fails, the text of the test is printed as part of the failure message. How the TEST function works is beyond the scope of this book, but it's useful to have around; you'll be using it for future test harnesses. Compare your test program with Listing 30-5.

Listing 30-5. Testing the `rational` Comparison Operators

```
#include <cassert>
#include <cstdlib>
#include <iostream>
#include "test.hpp"

... struct rational omitted for brevity ...

int main()
{
  rational a{60, 5};
  rational b{12, 1};
  rational c{-24, -2};
  TEST(a == b);
  TEST(a >= b);
  TEST(a <= b);
  TEST(b <= a);
  TEST(b >= a);
  TEST(b == c);
  TEST(b >= c);
  TEST(b <= c);
  TEST(a == c);
  TEST(a >= c);
  TEST(a <= c);

  rational d{109, 10};
  TEST(d < a);
  TEST(d <= a);
  TEST(d != a);
  TEST(a > d);
  TEST(a >= d);
  TEST(a != d);

  rational e{241, 20};
  TEST(e > a);
  TEST(e >= a);
  TEST(e != a);
  TEST(a < e);
  TEST(a <= e);
  TEST(a != e);
}
```

Arithmetic Operators

Comparison is fine, but arithmetic operators are much more interesting. You can overload any or all of the arithmetic operators. Binary operators take two parameters, and unary operators take one parameter. You can choose any return type that makes sense. Listing 30-6 shows the binary addition operator and the unary negation operator.

Listing 30-6. Addition Operator for the `rational` Type

```
rational operator+(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.denominator + rhs.numerator * lhs.denominator,
                  lhs.denominator * rhs.denominator};
}

rational operator-(rational const& r)
{
  return rational{-r.numerator, r.denominator};
}
```

Write the other arithmetic operators: -, *, and /. Ignore for the moment the issue of division by zero. Compare your functions with mine, which are presented in Listing 30-7.

Listing 30-7. Arithmetic Operators for the `rational` Type

```
rational operator-(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.denominator - rhs.numerator * lhs.denominator,
                  lhs.denominator * rhs.denominator};
}

rational operator*(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.numerator, lhs.denominator * rhs.denominator};
}

rational operator/(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.denominator, lhs.denominator * rhs.numerator};
}
```

Adding, subtracting, etc. with `rational` numbers is fine, but more interesting is the issue of mixing types. For example, what is the value of `3 * rational(1, 3)`? **Try it.** Collect the definition of the `rational` type with all the operators and write a `main()` function that computes that expression and stores it somewhere. Choose a type for the result variable that makes sense to you, then determine how best to print that value to `std::cout`.

Do you expect the expression to compile without errors? ________________

What is the result type of the expression? ________________

What value do you expect as the result? ________________

Explain your observations.

____

____

____

____

It turns out that `rational`'s one-argument constructor tells the compiler it can construct a `rational` from an `int` anytime it needs to do so. It does so automatically, so the compiler sees the integer `3` and a multiplication of an `int` and a `rational` object. It knows about `operator*` between two `rational`s, and it knows it cannot use the built-in `*` operator with a `rational` operand. Thus, the compiler decides its best response is to convert the `int` to a `rational` (by invoking `rational{3}`), and then it can apply the custom `operator*` that multiplies two `rational` objects, yielding a `rational` result, namely, `rational{1, 1}`. It does all this automatically on your behalf. Listing 30-8 illustrates one way to write the test program.

Listing 30-8. Test Program for Multiplying an Integer and a Rational Number

```
#include <cassert>
#include <cstdlib>
#include <iostream>

... struct rational omitted for brevity ...

int main()
{
  rational result{3 * rational{1, 3}};
  std::cout << result.numerator << '/' << result.denominator << '\n';
}
```

Being able to construct a `rational` object automatically from an `int` is a great convenience. You can easily write code that performs operations on integers and rational numbers without concerning yourself with type conversions all the time. You'll find this same convenience when mixing integers and floating-point numbers. For example, you can write `1+2.0` without having to perform a type cast: `static_cast<double>(1)+2.0`.

On the other hand, all this convenience can be too convenient. **Try to compile the following code sample and see what your compiler reports**.

```
int a(3.14); // which one is okay,
int b{3.14}; // and which is an error?
```

You have seen that parentheses are needed when initializing an `auto` declaration, and we use curly braces everywhere else. In truth, you can use parentheses for many of these initializations, but you pay a cost in safety. The compiler allows conversions that lose information, such as floating-point to integer, when you use parentheses for initialization, but it reports an error when you use curly braces.

The difference is critical for the `rational` type. Initializing a rational number using a floating-point number in parentheses truncates the number to an integer and uses the one-argument form of constructor. This isn't what you want at all. Instead, initializing `rational{3.14}` should produce the same result as `rational{314, 100}`.

Writing a high-quality conversion from floating-point to a fraction is beyond the scope of this book. Instead, let's just pick a reasonable power of 10 and use that as the denominator. Say we choose 100000, then `rational{3.14159}` would be treated as `rational{static_cast<int>(3.14159 * 10000), 10000}`. **Write the constructor for a floating-point number**. I recommend using a delegating constructor; that is, write the floating-point constructor so it invokes another constructor.

Compare your result with mine in Listing 30-9. A better solution uses `numeric_limits` to determine the number of decimal digits of precision `double` can support, and tries to preserve as much precision as possible. An even better solution uses the radix of the floating-point implementation, instead of working in base 10.

Listing 30-9. Constructing a Rational Number from a Floating-Point Argument

```
struct rational
{
  rational(int num, int den)
  : numerator{num}, denominator{den}
  {
    reduce();
  }

  rational(double r)
  : rational{static_cast<int>(r * 10000), 10000}
  {}

  ... omitted for brevity ...

  int numerator;
  int denominator;
};
```

If you want to optimize a particular function for a particular argument type, you can do that too, by taking advantage of ordinary function overloading. You'd better make sure it's worth the extra work, however. Remember that the `int` operand can be the right-hand or left-hand operand, so you will have to overload both forms of the function, as shown in Listing 30-10.

Listing 30-10. Optimizing Operators for a Specific Operand Type

```
rational operator*(rational const& rat, int mult)
{
  return rational{rat.numerator * mult, rat.denominator};
}

inline rational operator*(int mult, rational const& rat)
{
  return rat * mult;
}
```

In such a simple case, it's not worth the added trouble to avoid a little extra arithmetic. However, in more complicated situations, such as division, you may have to write such code.

Math Functions

The C++ standard library offers a number of mathematical functions, such as `std::abs`, which computes absolute values (as you have already guessed). The C++ standard prohibits you from overloading these standard functions to operate on custom types, but you can still write functions that perform similar operations. In Exploration 52, you'll learn about namespaces, which will enable you to use the real function name. Whenever you write a custom numeric type, you should consider which math functions you should provide. In this case, absolute value makes perfect sense. **Write an absolute value function that works with rational numbers. Call it absval**.

Your `absval` function should take a `rational` parameter by value and return a `rational` result. As with the arithmetic operators I wrote, you may opt to use call-by-reference for the parameter. If so, make sure you declare the reference to be `const`. Listing 30-11 shows my implementation of `absval`.

Listing 30-11. Computing the Absolute Value of a Rational Number

```
rational absval(rational const& r)
{
  return rational{std::abs(r.numerator), r.denominator};
}
```

That was easy. What about the other math functions, such as `sqrt`, for computing square roots? Most of the other functions are overloaded for floating-point arguments. If the compiler knew how to convert a rational number to a floating-point number automatically, you could simply pass a `rational` argument to any of the existing floating-point functions, with no further work. **So, which floating-point type should you use?** _______________.

This question has no easy answer. A reasonable first choice might be `double`, which is the "default" floating-point type (e.g., floating-point literals have type `double`). On the other hand, what if someone really wants the extra precision `long double` offers? Or what if that person doesn't need much precision and prefers to use `float`?

The solution is to abandon the possibility of automatic conversion to a floating-point type and instead offer three functions that explicitly compute the floating-point value of the rational number. **Write as_float, as_double, and as_long_double**. Each of these member functions computes and returns the floating-point approximation for the rational number. The function name identifies the return type. You will have to cast the numerator and denominator to the desired floating-point type using `static_cast`, as you learned in Exploration 25. Listing 30-12 shows how I wrote these functions, with a sample program that demonstrates their use.

Listing 30-12. Converting to Floating-Point Types

```
struct rational
{
  float as_float()
  {
    return static_cast<float>(numerator) / denominator;
  }

  double as_double()
  {
    return numerator / static_cast<double>(denominator);
  }

  long double as_long_double()
  {
    return static_cast<long double>(numerator) /
           static_cast<long double>(denominator);
  }

... omitted for brevity ...

};

int main()
{
  rational pi{355, 113};
  rational bmi{90*100*100, 180*180}; // Body-mass index of 90 kg, 180 cm
  double circumference{0}, radius{10};
  circumference = 2 * pi.as_double() * radius;
  std::cout << "circumference of circle with radius " << radius << " is about "
            << circumference << '\n';
  std::cout << "bmi = " << bmi.as_float() << '\n';
}
```

As you can see, if one argument to / (or any other arithmetic or comparison operator) is floating-point, the other operand is converted to match. You can cast both operands or just one or the other. Pick the style that suits you best and stick with it.

One more task would make it easier to write test programs: overloading the I/O operators. That is the topic for the next Exploration.

EXPLORATION 31

Custom I/O Operators

Wouldn't it be nice to be able to read and write rational numbers directly, for example, `std::cout << rational{355, 113}`? In fact, C++ has everything you need, although the job is a little trickier than perhaps it should be. This Exploration introduces some of the pieces you need to accomplish this.

Input Operator

The I/O operators are just like any other operators in C++, and you can overload them the way you overload any other operator. The input operator, also known as an *extractor* (because it extracts data from a stream), takes `std::istream&` as its first parameter. It must be a non-`const` reference, because the function will modify the stream object. The second parameter must also be a non-`const` reference, because you will store the input value there. By convention, the return type is `std::istream&`, and the return value is the first parameter. That lets you combine multiple input operations in a single expression. (Go back to Listing 17-3 for an example.)

The body of the function must do the work of reading the input stream, parsing the input, and deciding how to interpret that input. Proper error handling is difficult, but the basics are easy. Every stream has a state mask that keeps track of errors. Table 31-1 lists the available state flags (declared in `<ios>`).

Table 31-1. *I/O State Flags*

Flag	Description
`badbit`	Unrecoverable error
`eofbit`	End of file
`failbit`	Invalid input or output
`goodbit`	No errors

If the input is not valid, the input function sets `failbit` in the stream's error state. When the caller tests whether the stream is okay, it tests the error state. If `failbit` is set, the check fails. (The test also fails if an unrecoverable error occurs, such as a hardware malfunction, but that's not pertinent to the current topic.)

Now you have to decide on a format for rational numbers. The format should be one that is flexible enough for a human to read and write easily but simple enough for a function to read and parse quickly. The input format must be able to read the output format and might be able to read other formats too.

Let's define the format as an integer, a slash (`/`), and another integer. White space can appear before or after any of these elements, unless the white space flag is disabled in the input stream. If the input contains an integer that is not followed by a slash, the integer becomes the resulting value (that is, the implicit denominator is 1). The input operator has to "unread" the character, which may be important to the rest of the program. The `unget()` member function does exactly that. The input operator for integers will do the same thing: read as many characters as possible until reading a character that is not part of the integer, then `unget` that last character.

Putting all these pieces together requires a little care, but is not all that difficult. Listing 31-1 presents the input operator. Add this operator to the rest of the `rational` type that you wrote in Exploration 30.

Listing 31-1. Input Operator

```
#include <ios>      // declares failbit, etc.
#include <istream>  // declares std::istream and the necessary >> operators

std::istream& operator>>(std::istream& in, rational& rat)
{
  int n{0}, d{0};
  char sep{'\0'};
  if (not (in >> n >> sep))
    // Error reading the numerator or the separator character.
    in.setstate(std::cin.failbit);
  else if (sep != '/')
  {
    // Read numerator successfully, but it is not followed by /.
    // Push sep back into the input stream, so the next input operation
    // will read it.
    in.unget();
    rat.assign(n, 1);
  }
  else if (in >> d)
    // Successfully read numerator, separator, and denominator.
    rat.assign(n, d);
  else
    // Error reading denominator.
    in.setstate(std::cin.failbit);

  return in;
}
```

Notice how `rat` is not modified until the function has successfully read both the numerator and the denominator from the stream. The goal is to ensure that if the stream enters an error state, the function does not alter `rat`.

The input stream automatically handles white space. By default, the input stream skips leading white space in each input operation, which means the `rational` input operator skips white space before the numerator, the slash separator, and the denominator. If the program turns off the `ws` flag, the input stream does not skip white space, and all three parts must be contiguous.

Output Operator

Writing the output operator, or *inserter* (so named because it inserts text into the output stream), has a number of hurdles, due to the plethora of formatting flags. You want to heed the desired field width and alignment, and you have to insert fill characters, as needed. Like any other output operator, you want to reset the field width but not change any other format settings.

One way to write a complicated output operator is to use a temporary output stream that stores its text in a `string`. The `std::ostringstream` type is declared in the `<sstream>` header. Use `ostringstream` the way you would use any other output stream, such as `cout`. When you are done, the `str()` member function returns the finished `string`.

To write the output operator for a `rational` number, create an `ostringstream`, and then write the numerator, separator, and denominator. Next, write the resulting string to the actual output stream. Let the stream itself handle the width, alignment, and fill issues when it writes the string. If you had written the numerator, slash, and denominator directly to the output stream, the width would apply only to the numerator, and the alignment would be wrong. Similar to an input operator, the first parameter has type `std::ostream&`, which is also the return type. The return value is the first parameter. The second parameter can use call-by-value, or you can pass a reference to `const`, as you can see in Listing 31-2. Add this code to Listing 31-1 and the rest of the rational type that you are defining.

Listing 31-2. Output Operator

```
#include <ostream>  // declares the necessary << operators
#include <sstream>  // declares the std::ostringstream type

std::ostream& operator<<(std::ostream& out, rational const& rat)
{
  std::ostringstream tmp{};
  tmp << rat.numerator;
  if (rat.denominator != 1)
    tmp << '/' << rat.denominator;
  out << tmp.str();

  return out;
}
```

Error State

The next step is to write a test program. Ideally, the test program should be able to continue when it encounters an invalid-input error. So now is a good time to take a closer look at how an I/O stream keeps track of errors.

As you learned earlier in this Exploration, every stream has a mask of error flags (see Table 31-1). You can test these flags, set them, or clear them. Testing the flags is a little unusual, however, so pay attention.

The way most programs in this book test for error conditions on a stream is to use the stream itself or an input operation as a condition. As you learned, an input operator function returns the stream, so these two approaches are equivalent. A stream converts to a `bool` result by returning the inverse of its `fail()` function, which returns `true`, if `failbit` or `badbit` are set.

In the normal course of an input loop, the program progresses until the input stream is exhausted. The stream sets `eofbit` when it reaches the end of the input stream. The stream's state is still good, in that `fail()` returns false, so the loop continues. However, the next time you try to read from the stream, it sees that no more input is available, sets `failbit`, and returns an error condition. The loop condition is `false`, and the loop exits.

The loop might also exit if the stream contains invalid input, such as non-numeric characters for integer input, or the loop can exit if there is a hardware error on the input stream (such as a disk failure). Until now, the programs in this book didn't bother to test why the loop exited. To write a good test program, however, you have to know the cause.

First, you can test for a hardware or similar error by calling the `bad()` member function, which returns true if `badbit` is set. That means something terrible happened to the file, and the program can't do anything to fix the problem.

Next, test for normal end-of-file by calling the `eof()` member function, which is `true` only when `eofbit` is set. If `bad()` and `eof()` are both `false` and `fail()` is `true`, this means the stream contains invalid input. How your program should handle an input failure depends on your particular circumstances. Some programs must exit immediately; others may try to continue. For example, your test program can reset the error state by calling the `clear()` member function, then continue running. After an input failure, you may not know the stream's position, so you don't know what the stream is prepared to read next. This simple test program skips to the next line.

Listing 31-3 demonstrates a test program that loops until end-of-file or an unrecoverable error occurs. If the problem is merely invalid input, the error state is cleared, and the loop continues.

Listing 31-3. Testing the I/O Operators

```
... omitted for brevity ...

/// Tests for failbit only
bool iofailure(std::istream& in)
{
  return in.fail() and not in.bad();
}

int main()
{
  rational r{0};

  while (std::cin)
  {
    if (std::cin >> r)
      // Read succeeded, so no failure state is set in the stream.
      std::cout << r << '\n';
    else if (iofailure(std::cin))
    {
      // Only failbit is set, meaning invalid input. Clear the state,
      // and then skip the rest of the input line.
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<int>::max(), '\n');
    }
  }

  if (std::cin.bad())
    std::cerr << "Unrecoverable input failure\n";
}
```

The `rational` type is nearly finished. The next Exploration tackles assignment operators and seeks to improve the constructors.

EXPLORATION 32

Assignment and Initialization

The final step needed to complete this stage of the `rational` type is to write assignment operators and to improve the constructors. It turns out C++ does some work for you, but you often want to fine-tune that work. Let's find out how.

Assignment Operator

Until now, all the `rational` operators have been free functions. The assignment operator is different. The C++ standard requires that it be a member function. One way to write this function is shown in Listing 32-1.

Listing 32-1. First Version of the Assignment Operator

```
struct rational
{
  rational(int num, int den)
  : numerator{num}, denominator{den}
  {
    reduce();
  }

  rational& operator=(rational const& rhs)
  {
    numerator = rhs.numerator;
    denominator = rhs.denominator;
    reduce();
    return *this;
  }
  int numerator;
  int denominator;
};
```

Several points require further explanation. When you implement an operator as a free function, you need one parameter per operand. Thus, binary operators require a two-parameter function, and unary operators require a one-parameter function. Member functions are different, because the object itself is an operand (always the left-hand operand), and the object is implicitly available to all member functions; therefore, you need one fewer parameter. Binary operators require a single parameter (as you can see in Listing 32-1), and unary operators require no parameters (examples to follow).

The convention for assignment operators is to return a reference to the enclosing type. The value to return is the object itself. You can obtain the object with the expression `*this` (`this` is a reserved keyword).

Because `*this` is the object itself, another way to refer to members is to use the dot operator (e.g., `(*this).numerator`) instead of the basic `numerator`. Another way to write `(*this).numerator` is `this->numerator`. The meaning is the same; the alternative syntax is mostly a convenience. Writing `this->` is not necessary for these simple functions, but it's often a good idea. When you read a member function, and you have trouble discerning the members from the nonmembers, that's a signal that you have to help the reader by using `this->` before all the member names. Listing 32-2 shows the assignment operator with explicit use of `this->`.

Listing 32-2. Assignment Operator with Explicit Use of `this->`

```
rational& operator=(rational const& that)
{
  this->numerator = that.numerator;
  this->denominator = that.denominator;
  reduce();
  return *this;
}
```

The right-hand operand can be anything you want it to be. For example, you may want to optimize assignment of an integer to a `rational` object. The way the assignment operator works with the compiler's rules for automatic conversion, the compiler treats such an assignment (e.g., `r = 3`) as an implicit construction of a temporary `rational` object, followed by an assignment of one `rational` object to another.

Write an assignment operator that takes an `int` parameter. Compare your solution with mine, which is shown in Listing 32-3.

Listing 32-3. Assignment of an Integer to a `rational`

```
rational& operator=(int num)
{
  this->numerator = num;
  this->denominator = 1; // no need to call reduce()
  return *this;
}
```

If you do not write an assignment operator, the compiler writes one for you. In the case of the simple `rational` type, it turns out that the compiler writes one that works exactly like the one in Listing 32-2, so there was actually no need to write it yourself (except for instructional purposes). When the compiler writes code for you, it is hard for the human reader to know which functions are actually defined. Also, it is harder to document the implicit functions. So C++ 11 lets you state explicitly that you want the compiler to supply a special function for you, by following a declaration (not a definition) with `=default` instead of a function body.

```
rational& operator=(rational const&) = default;
```

Constructors

The compiler also writes a constructor automatically, specifically one that constructs a `rational` object by copying all the data members from another `rational` object. This is called a *copy constructor*. Any time you pass a `rational` argument by value to a function, the compiler uses the copy constructor to copy the argument value to the parameter. Any time you define a `rational` variable and initialize it with the value of another `rational` value, the compiler constructs the variable by calling the copy constructor.

As with the assignment operator, the compiler's default implementation is exactly what we would write ourselves, so there is no need to write the copy constructor. As with an assignment operator, you can state explicitly that you want the compiler to supply its default copy constructor.

```
rational(rational const&) = default;
```

If you don't write any constructors for a type, the compiler also creates a constructor that takes no arguments, called a *default constructor*. The compiler uses the default constructor when you define a variable of a custom type and do not provide an initializer for it. The compiler's implementation of the default constructor merely calls the default constructor for each data member. If a data member has a built-in type, the member is left uninitialized. In other words, if we did not write any constructors for `rational`, any `rational` variable would be uninitialized, so its numerator and denominator would contain garbage values. That's bad—very bad. All the operators assume the `rational` object has been reduced to normal form. They would fail if you passed an uninitialized `rational` object to them. The solution is simple: don't let the compiler write its default constructor. Instead, you write one.

All you have to do is write any constructor at all. This will prevent the compiler from writing its own default constructor. (It will still write its own copy constructor.)

Early on, we wrote a constructor for the `rational` type, but it was not a default constructor. As a result, you could not define a `rational` variable and leave it uninitialized or initialize it with empty braces. (You may have run into that issue when writing your own test program.) Uninitialized data is a bad idea, and having default constructors is a good idea. So write a default constructor to make sure a `rational` variable that has no initializer has a well-defined value nonetheless. What value should you use? I recommend zero, which is in keeping with the spirit of the default constructors for types such as `string` and `vector`. **Write a default constructor for `rational` to initialize the value to zero**.

Compare your solution with mine, which is presented in Listing 32-4.

Listing 32-4. Overloaded Constructors for `rational`

```
rational()
: numerator{0}, denominator{1}
{}
```

Putting It All Together

Before we take leave of the `rational` type (only temporarily; we will return), let's put all the pieces together, so you can see what you've accomplished over the past four Explorations. Listing 32-5 shows the complete definition of `rational` and the related operators.

Listing 32-5. Complete Definition of `rational` and Its Operators

```
#include <cassert>
#include <cstdlib>
#include <istream>
#include <ostream>
#include <sstream>

/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
int gcd(int n, int m)
{
  n = std::abs(n);
  while (m != 0) {
    int tmp(n % m);
    n = m;
    m = tmp;
  }
  return n;
}

/// Represent a rational number (fraction) as a numerator and denominator.
struct rational
{
  rational()
  : numerator{0}, denominator{1}
  {/*empty*/}

  rational(int num)
  : numerator{num}, denominator{1}
  {/*empty*/}

  rational(int num, int den)
  : numerator{num}, denominator{den}
  {
    reduce();
  }

  rational(double r)
  : rational{static_cast<int>(r * 10000), 10000}
  {/*empty*/}

  rational& operator=(rational const& that)
  {
    this->numerator = that.numerator;
    this->denominator = that.denominator;
    reduce();
    return *this;
  }

  float as_float()
  {
    return static_cast<float>(numerator) / denominator;
  }

  double as_double()
  {
    return static_cast<double>(numerator) / denominator;
  }

  long double as_long_double()
  {
    return static_cast<long double>(numerator) /
           denominator;
  }

  /// Assign a numerator and a denominator, then reduce to normal form.
  void assign(int num, int den)
  {
    numerator = num;
    denominator = den;
    reduce();
  }

  /// Reduce the numerator and denominator by their GCD.
  void reduce()
  {
    assert(denominator != 0);
    if (denominator < 0)
    {
      denominator = -denominator;
      numerator = -numerator;
    }
    int div{gcd(numerator, denominator)};
    numerator = numerator / div;
    denominator = denominator / div;
  }

  int numerator;
  int denominator;
};

/// Absolute value of a rational number.
rational abs(rational const& r)
{
  return rational{std::abs(r.numerator), r.denominator};
}

/// Unary negation of a rational number.
rational operator-(rational const& r)
{
  return rational{-r.numerator, r.denominator};
}

/// Add rational numbers.
rational operator+(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.denominator + rhs.numerator * lhs.denominator,
                  lhs.denominator * rhs.denominator};
}

/// Subtraction of rational numbers.
rational operator-(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.denominator - rhs.numerator * lhs.denominator,
                  lhs.denominator * rhs.denominator};
}
```

```
/// Multiplication of rational numbers.
rational operator*(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.numerator, lhs.denominator * rhs.denominator};
}

/// Division of rational numbers.
/// TODO: check for division-by-zero
rational operator/(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator * rhs.denominator, lhs.denominator * rhs.numerator};
}

/// Compare two rational numbers for equality.
bool operator==(rational const& a, rational const& b)
{
  return a.numerator == b.numerator and a.denominator == b.denominator;
}

/// Compare two rational numbers for inequality.
inline bool operator!=(rational const& a, rational const& b)
{
  return not (a == b);
}
/// Compare two rational numbers for less-than.
bool operator<(rational const& a, rational const& b)
{
  return a.numerator * b.denominator < b.numerator * a.denominator;
}

/// Compare two rational numbers for less-than-or-equal.
inline bool operator<=(rational const& a, rational const& b)
{
  return not (b < a);
}
/// Compare two rational numbers for greater-than.
inline bool operator>(rational const& a, rational const& b)
{
  return b < a;
}

/// Compare two rational numbers for greater-than-or-equal.
inline bool operator>=(rational const& a, rational const& b)
{
  return not (b > a);
}

/// Read a rational number.
/// Format is @em integer @c / @em integer.
std::istream& operator>>(std::istream& in, rational& rat)
```

```
{
  int n{0}, d{0};
  char sep{'\0'};
  if (not (in >> n >> sep))
    // Error reading the numerator or the separator character.
    in.setstate(in.failbit);
  else if (sep != '/')
  {
    // Push sep back into the input stream, so the next input operation
    // will read it.
    in.unget();
    rat.assign(n, 1);
  }
  else if (in >> d)
    // Successfully read numerator, separator, and denominator.
    rat.assign(n, d);
  else
    // Error reading denominator.
    in.setstate(in.failbit);

  return in;
}

/// Write a rational numbers.
/// Format is @em numerator @c / @em denominator.
std::ostream& operator<<(std::ostream& out, rational const& rat)
{
  std::ostringstream tmp{};
  tmp << rat.numerator << '/' << rat.denominator;
  out << tmp.str();

  return out;
}
```

I encourage you to add tests to the program in Listing 30-5, to exercise all the latest features of the `rational` class. Make sure everything works the way you expect it. Then put aside `rational` for the next Exploration, which takes a closer look at the foundations of writing custom types.

EXPLORATION 33

Writing Classes

The `rational` type is an example of a *class*. Now that you've seen a concrete example of writing your own class, it's time to understand the general rules that govern all classes. This Exploration and the next four lay the foundation for this important aspect of C++ programming.

Anatomy of a Class

A class has a name and *members*—data members, member functions, and even member typedefs and nested classes. You start a class definition with the `struct` keyword. (You might wonder why you would not start a class definition with the `class` keyword. Please be patient; all will become clear in Exploration 35.) Use curly braces to surround the body of the class definition, and the definition ends with a semicolon. Within the curly braces, you list all the members. Declare data members in a manner similar to a local variable definition. You write member functions in the same manner as you would a free function. Listing 33-1 shows a simple class definition that contains only data members.

Listing 33-1. Class Definition for a Cartesian Point

```
struct point
{
  double x;
  double y;
};
```

Listing 33-2 demonstrates how C++ lets you list multiple data members in a single declaration. Except for trivial classes, this style is uncommon. I prefer to list each member separately, so I can include a comment explaining the member, what it's used for, what constraints apply to it, and so on. Even without the comment, a little extra clarity goes a long way.

Listing 33-2. Multiple Data Members in One Declaration

```
struct point
{
  double x, y;
};
```

As with any other name in a C++ source file, before you can use a class name, the compiler must see its declaration or definition. You can use the name of a class within its own definition.

Use the class name as a type name, to define local variables, function parameters, function return types, and even other data members. The compiler knows about the class name from the very start of the class definition, so you can use its name as a type name inside the class definition.

When you define a variable using a class type, the compiler sets aside enough memory so the variable can store its own copy of every data member of the class. For example, define an object with type `point`, and the object contains the `x` and `y` members. Define another object of type `point`, and that object contains its own, separate `x` and `y` members.

Use the dot (`.`) operator to access the members, as you have been doing throughout this book. The object is the left-hand operand, and the member name is the right-hand operand, as shown in Listing 33-3.

Listing 33-3. Using a Class and Its Members

```
#include <iostream>

struct point
{
  double x;
  double y;
};

int main()
{
  point origin{}, unity{};
  origin.x = 0;
  origin.y = 0;
  unity.x = 1;
  unity.y = 1;
  std::cout << "origin = (" << origin.x << ", " << origin.y << ")\n";
  std::cout << "unity  = (" << unity.x  << ", " << unity.y  << ")\n";
}
```

Member Functions

In addition to data members, you can have member functions. Member function definitions look very similar to ordinary function definitions, except you define them as part of a class definition. Also, a member function can call other member functions of the same class and can access data members of the same class. Listing 33-4 shows some member functions added to class `point`.

Listing 33-4. Member Functions for Class `point`

```
#include <cmath> // for sqrt and atan2

struct point
{
  /// Distance to the origin.
  double distance()
  {
    return std::sqrt(x*x + y*y);
  }
  /// Angle relative to x-axis.
  double angle()
  {
    return std::atan2(y, x);
  }
  /// Add an offset to x and y.
  void offset(double off)
  {
    offset(off, off);
  }
  /// Add an offset to x and an offset to y
  void offset(double  xoff, double yoff)
  {
    x = x + xoff;
    y = y + yoff;
  }

  /// Scale x and y.
  void scale(double mult)
  {
    this->scale(mult, mult);
  }
  /// Scale x and y.
  void scale(double xmult, double ymult)
  {
    this->x = this->x * xmult;
    this->y = this->y * ymult;
  }
  double x;
  double y;
};
```

For each member function, the compiler generates a hidden parameter named `this`. When you call a member function, the compiler passes the object as the hidden argument. In a member function, you can access the object with the expression `*this`. The C++ syntax rules specify that the member operator (`.`) has higher precedence than the * operator, so you need parentheses around `*this` (e.g., `(*this).x`). As a syntactic convenience, another way to write the same expression is `this->x`, several examples of which you can see in Listing 33-4.

The compiler is smart enough to know when you use a member name, so the use of `this->` is optional. If a name has no local definition, and it is the name of a member, the compiler assumes you want to use the member. Some programmers prefer to always include `this->` for the sake of clarity—in a large program, you can easily lose track of which names are member names. Other programmers find the extra `this->` to be clutter and use it only when necessary. My recommendation is the latter. You need to learn to read C++ classes, and one of the necessary skills is to be able to read a class definition, find the member names, and keep track of those names while you read the class definition.

A number of programmers employ a more subtle technique, which involves using a special prefix or suffix to denote data member names. For example, a common technique is to use the prefix `m_` for all data members ("m" is short for *member*). Another common technique is a little less intrusive: using a plain underscore (_) suffix. I prefer a suffix to a prefix, because suffixes interfere less than prefixes, so they don't obscure the important part of a name. From now on, I will adopt the practice of appending an underscore to every data member name.

NO LEADING UNDERSCORE

If you want to use only an underscore to denote members, use it as a suffix, not a prefix. The C++ standard sets aside certain names and prohibits you from using them. The actual rules are somewhat lengthy, because C++ inherits a number of restrictions from the C standard library. For example, you should not use any name that begins with `E` and is followed by a digit or an uppercase letter. (That rule seems arcane, but the C standard library defines several error code names, such as `ERANGE`, for a range error in a math function. This rule lets the library add new names in the future and lets those who implement libraries add vendor-specific names.)

I like simplicity, so I follow three basic rules. These rules are slightly more restrictive than the official C++ rules, but not in any burdensome way:

- Do not use any name that contains two consecutive underscores (`like__this`).
- Do not use any name that starts with an underscore (`_like_this`).
- Do not use any name that is all uppercase (`LIKE_THIS`).

Using a reserved name results in undefined behavior. The compiler may not complain, but the results are unpredictable. Typically, a standard library implementation must invent many additional names for its internal use. By defining certain names that the application programmer cannot use, C++ ensures the library-writer can use these names within the library. If you accidentally use a name that conflicts with an internal library name, the result could be chaos or merely a subtle shift in a function's implementation.

Constructor

As you learned in Exploration 29, a constructor is a special member function that initializes an object's data members. You saw several variations on how to write a constructor, and now it's time to learn a few more.

When you declare a data member, you can also provide an initializer. The initializer is a default value that the compiler uses in case a constructor does not initialize that member. Use the normal initialization syntax of providing a value or values in curly braces.

```
struct point {
  int x{1};
  int y{};
  point() {} // initializes x to 1 and y to 0
};
```

Use this style of initializing data members only when a particular member needs a single value in all or nearly all constructors. By separating the initial value from the constructor, it makes the constructor harder to read and understand. The human reader must read the constructor and the data member declarations to know how the object is initialized. On the other hand, using default initializers is a great way to ensure that data members of built-in type, such as `int`, are always initialized.

Recall that constructors can be overloaded, and the compiler chooses which constructor to call, based on the arguments in the initializer. I like to use curly braces to initialize an object. The values in the curly braces are passed to the constructor in the same manner as function arguments to an ordinary function. In fact, C++ 03 used parentheses to initialize an object, so an initializer looked very much like a function call. C++ 11 still allows this style of initializer, and you saw that when using an `auto` declaration, you must use parentheses. But in all other cases, curly braces are better. Exploration 30 demonstrated that curly braces provide greater type safety.

Another key difference with curly braces is that you can initialize a container, such as a `vector`, with a series of values in curly braces, as follows:

```
std::vector<int> data{ 1, 2, 3 };
```

This introduces a problem. The vector type has several constructors. For example, a two-argument constructor lets you initialize a vector with many copies of a single value. A vector with ten zeroes, for example, can be initialized as follows:

```
std::vector<int> ten_zeroes(10, 0);
```

Note that I used parentheses. What if I used curly braces? **Try it. What happens?**

____

____

The vector is initialized with two integers: 10 and 0. The rule is that containers treat curly braces as a series of values with which to initialize the container contents. Curly braces can be used in a few other cases, such as copying a container, but in general, you have to use parentheses to call any of the other constructors.

Write a constructor almost the same way you would an ordinary member function, but with a few differences.

- Omit the return type.
- Use plain `return;` (return statements that do not return values).
- Use the class name as the function name.
- Add an initializer list after a colon to initialize the data members. An initializer can also invoke another constructor, passing arguments to that constructor. Delegating construction to a common constructor is a great way to ensure rules are properly enforced by all constructors.

Listing 33-5 shows several examples of constructors added to class `point`.

Listing 33-5. Constructors for Class `point`

```
struct point
{
  point()
  : point{0.0, 0.0}
  {}
  point(double x, double y)
  : x_{x}, y_{y}
  {}
  point(point const& pt)
  : point{pt.x_, pt.y_}
  {}
  double x_;
  double y_;
};
```

Initialization is one of the key differences between class types and built-in types. If you define an object of built-in type without an initializer, you get a garbage value, but objects of class type are always initialized by calling a constructor. You always get a chance to initialize the object's data members. The difference between built-in types and class types are also evident in the rules C++ uses to initialize data members in a constructor.

A constructor's initializer list is optional, but I recommend you always provide it, unless every data member has an initializer. The initializer list appears after a colon, which follows the closing parenthesis of the constructor's parameter list; it initializes each data member in the same order in which you declare them in the class definition, ignoring the order in the initializer list. To avoid confusion, always write the initializer list in the same order as the data members. Member initializers are separated by commas and can spill onto as many lines as you need. Each member initializer provides the initial value of a single data member or uses the class name to invoke another constructor. List the member name, followed by its initializer in curly braces. Initializing a data member is the same as initializing a variable and follows the same rules.

If you don't write any constructors for your class, the compiler writes its own default constructor. The compiler's default constructor is just like a constructor that omits an initializer list.

```
struct point {
  point() {} // x_ is initialized to 0, and y_ is uninitialized
  double x_{};
  double y_;
};
```

When the compiler writes a constructor for you, the constructor is *implicit*. If you write any constructor, the compiler suppresses the implicit default constructor. If you want a default constructor in that case, you must write it yourself.

In some applications, you may want to avoid the overhead of initializing the data members of `point`, because your application will immediately assign a new value to the `point` object. Most of the time, however, caution is best.

A *copy constructor* is one that takes a single argument of the same type as the class, passed by reference. The compiler automatically generates calls to the copy constructor when you pass objects by value to functions, or when functions return objects. You can also initialize a `point` object with the value of another `point` object, and the compiler generates code to invoke the copy constructor.

```
point pt1;          // default constructor
point p2{pt1};      // copy constructor
```

If you don't write your own copy constructor, the compiler writes one for you. The automatic copy constructor calls the copy constructor for every data member, just like the one in Listing 33-5. Because I wrote one that is exactly like the one the compiler writes implicitly, there is no reason to write it explicitly. Let the compiler do its job.

To help you visualize how the compiler calls constructors, read Listing 33-6. Notice how it prints a message for each constructor use.

Listing 33-6. Visual Constructors

```
#include <iostream>

struct demo
{
  demo()      : demo{0} { std::cout << "default constructor\n"; }
  demo(int x) : x_{x} { std::cout << "constructor(" << x << ")\n"; }
  demo(demo const& that)
  : x_{that.x_}
  {
    std::cout << "copy constructor(" << x_ << ")\n";
  }
  int x_;
};

demo addone(demo d)
{
  ++d.x_;
  return d;
}

int main()
{
  demo d1{};
  demo d2{d1};
  demo d3{42};
  demo d4{addone(d3)};
}
```

Predict the output from running the program in Listing 33-6.

____

____

____

____

____

____

____

Check your prediction. **Were you correct?** ________________

The compiler is allowed to perform some minor optimizations when passing arguments to functions and accepting return values. For example, instead of copying a demo object to the addone return value, and then copying the return value to initialize d4, the C++ standard permits compilers to remove unnecessary calls to the copy constructor. Not all compilers perform this optimization, and not all do so in the same manner. Most compilers require a command line switch or project option to be set before it optimizes. Thus, the exact number of calls to the copy constructor can vary slightly from one compiler or platform to another, or from one set of command line switches to another. When I run the program, I get the following:

```
constructor(0)
default constructor
copy constructor(0)
constructor(42)
copy constructor(42)
copy constructor(43)
```

Defaulted and Deleted Constructors

If you do not supply any constructors, the compiler implicitly writes a default constructor and a copy constructor. If you write at least one constructor of any variety, the compiler does not implicitly write a default constructor, but it still gives you a copy constructor if you don't write one yourself.

You can take control over the compiler's implicit behavior without writing any of your own constructors. Write a function header without a body for the constructor, and use `=default` to get the compiler's implicit definition. Use `=delete` to suppress that function. For example, if you don't want anyone creating copies of a class, note the following:

```
struct dont_copy
{
   dont_copy(dont_copy const&) = delete;
};
```

More common is letting the compiler write its copy constructor but telling the human reader explicitly. As you learn more about C++, you will learn that the rules for which constructors the compiler writes for you, and when it writes them, are more complicated than what I've presented so far. I urge you to get into the habit of stating when you ask the compiler to implicitly provide a constructor, even if it seems trivially obvious.

```
struct point
{
  point() = default;
  point(point const&) = default;
  int x, y;
};
```

That was easy. The next Exploration starts with a real challenge.

EXPLORATION 34

More About Member Functions

Member functions and constructors are even more fun than what you've learned so far. This Exploration continues to uncover their mysteries.

Revisiting Project 1

What did you find most frustrating about Project 1 (Exploration 28)? If you are anything like me (although I hope you're not, for your own sake), you may have been disappointed that you had to define several separate vectors to store one set of records. However, without knowing about classes, that was the only feasible approach. Now that you've been introduced to classes, you can fix the program. **Write a class definition to store one record.** Refer back to Exploration 28 for details. To summarize, each record keeps track of an integer height in centimeters, an integer weight in kilograms, the calculated BMI (which you can round off to an integer), the person's sex (letter `'M'` or `'F'`), and the person's name (a `string`).

Next, **write a read member function that reads a single record from an `istream`**. It takes two arguments: an `istream` and an integer. Prompt the user for each piece of information by writing to `std::cout`. The integer argument is the record number, which you can use in the prompts. **Write a print member function that prints one record**; it takes an `ostream` and an integer threshold as arguments.

Finally, modify the program to take advantage of the new class you wrote. Compare your solution to mine, shown in Listing 34-1.

Listing 34-1. New BMI Program

```
#include <algorithm>
#include <cstdlib>
#include <iomanip>
#include <iostream>
#include <limits>
#include <locale>
#include <string>
#include <vector>

/// Compute body-mass index from height in centimeters and weight in kilograms.
int compute_bmi(int height, int weight)
{
   return static_cast<int>(weight * 10000 / (height * height) + 0.5);
}

/// Skip the rest of the input line.
void skip_line(std::istream& in)
```

```
{
  in.ignore(std::numeric_limits<int>::max(), '\n');
}

/// Represent one person's record, storing the person's name, height, weight,
/// sex, and body-mass index (BMI), which is computed from the height and weight.
struct record
{
  record() : height_{0}, weight_{0}, bmi_{0}, sex_{'?'}, name_{}
  {}

  /// Get this record, overwriting the data members.
  /// Error-checking omitted for brevity.
  /// @return true for success or false for eof or input failure
  bool read(std::istream& in, int num)
  {
    std::cout << "Name " << num << ": ";
    std::string name{};
    if (not std::getline(in, name))
      return false;

    std::cout << "Height (cm): ";
    int height{};
    if (not (in >> height))
      return false;
    skip_line(in);

    std::cout << "Weight (kg): ";
    int weight{};
    if (not (in >> weight))
      return false;
    skip_line(in);

    std::cout << "Sex (M or F): ";
    char sex{};
    if (not (in >> sex))
      return false;
    skip_line(in);
    sex = std::toupper(sex, std::locale());

    // Store information into data members only after reading
    // everything successfully.
    name_ = name;
    height_ = height;
    weight_ = weight;
    sex_ = sex;
    bmi_ = compute_bmi(height_, weight_);
    return true;
  }
  /// Print this record to @p out.
  void print(std::ostream& out, int threshold)
  {
    out << std::setw(6) << height_
        << std::setw(7) << weight_
        << std::setw(3) << sex_
        << std::setw(6) << bmi_;
    if (bmi_ >= threshold)
      out << '*';
    else
      out << ' ';
    out << ' ' << name_ << '\n';
  }

  int height_;       ///< height in centimeters
  int weight_;       ///< weight in kilograms
  int bmi_;          ///< Body-mass index
  char sex_;         ///< 'M' for male or 'F' for female
  std::string name_; ///< Person's name
};


/** Print a table.
 * Print a table of height, weight, sex, BMI, and name.
 * Print only records for which sex matches @p sex.
 * At the end of each table, print the mean and median BMI.
 */
void print_table(char sex, std::vector<record>& records, int threshold)
{
  std::cout << "Ht(cm) Wt(kg) Sex  BMI  Name\n";

  float bmi_sum{};
  long int bmi_count{};
  std::vector<int> tmpbmis{}; // store only the BMIs that are printed
                              // in order to compute the median
  for (auto rec : records)
  {
    if (rec.sex_ == sex)
    {
      bmi_sum = bmi_sum + rec.bmi_;
      ++bmi_count;
      tmpbmis.push_back(rec.bmi_);
      rec.print(std::cout, threshold);
    }
  }

  // If the vectors are not empty, print basic statistics.
  if (bmi_count != 0)
  {
    std::cout << "Mean BMI = "
              << std::setprecision(1) << std::fixed << bmi_sum / bmi_count
              << '\n';
    // Median BMI is trickier. The easy way is to sort the
    // vector and pick out the middle item or items.
    std::sort(tmpbmis.begin(), tmpbmis.end());
    std::cout << "Median BMI = ";
    // Index of median item.
    std::size_t i{tmpbmis.size() / 2};
    if (tmpbmis.size() % 2 == 0)
      std::cout << (tmpbmis.at(i) + tmpbmis.at(i-1)) / 2.0 << '\n';
    else
      std::cout << tmpbmis.at(i) << '\n';
  }
}

/** Main program to compute BMI. */
int main()
{
  std::locale::global(std::locale{""});
  std::cout.imbue(std::locale{});
  std::cin.imbue(std::locale{});

  std::vector<record> records{};
  int threshold{};

  std::cout << "Enter threshold BMI: ";
  if (not (std::cin >> threshold))
    return EXIT_FAILURE;
  skip_line(std::cin);

  std::cout << "Enter name, height (in cm),"
               " and weight (in kg) for each person:\n";
  record rec{};
  while (rec.read(std::cin, records.size()+1))
  {
    records.push_back(rec);
    std::cout << "BMI = " << rec.bmi_ << '\n';
  }

  // Print the data.
  std::cout << "\n\nMale data\n";
  print_table('M', records, threshold);
  std::cout << "\nFemale data\n";
  print_table('F', records, threshold);
}
```

That's a lot to swallow, so take your time. I'll wait here until you're done. When faced with a new class that you have to read and understand, start by reading the comments (if any). One approach is to first skim lightly over the class to identify the members (function and data), then reread the class to understand the member functions in depth. Tackle one member function at a time.

You may be asking yourself why I didn't overload the `>>` and `<<` operators to read and write `record` objects. The requirements of the program are a little more complicated than what these operators offer. For example, reading a `record` also involves printing prompts, and each prompt includes an ordinal, so the user knows which record to type.

Some records are printed differently than others, depending on the threshold. The `>>` operator has no convenient way to specify the threshold. Overloading I/O operators is great for simple types but usually is not appropriate for more complicated situations.

Const Member Functions

Take a closer look at the `print_table` function. Notice anything unusual or suspicious about its parameters? The `records` argument is passed by reference, but the function never modifies it, so you really should pass it as a reference to `const`. Go ahead and make that change. **What happens?**

____

____

You should see an error from the compiler. When `records` is `const`, the `auto rec : records` type must declare `rec` as `const` too. Thus, when `print_table` calls `rec.print()`, inside the `print()` function, this refers to a `const record` object. Although `print()` does not modify the `record` object, it could, and the compiler must allow for the possibility. You have to tell the compiler that `print()` is safe and doesn't modify any data members. Do so by adding a `const` modifier between the `print()` function signature and the function body. Listing 34-2 shows the new definition of the `print` member function.

Listing 34-2. Adding the `const` Modifier to `print`

```
  /// Print this record to @p out.
  void print(std::ostream& out, int threshold)
  const
  {
    out << std::setw(6) << height_
        << std::setw(7) << weight_
        << std::setw(3) << sex_
        << std::setw(6) << bmi_;
    if (bmi_ >= threshold)
      out << '*';
    else
      out << ' ';
    out << ' ' << name_ << '\n';
  }
```

As a general rule, use the `const` modifier for any member function that does not change any data members. This ensures that you can call the member function when you have a `const` object. **Copy the code from Listing 33-4 and modify it to add `const` modifiers where appropriate.** Compare your result with mine in Listing 34-3.

Listing 34-3. `const` Member Functions for Class `point`

```
#include <cmath> // for sqrt and atan2

struct point
{
  /// Distance to the origin.
  double distance()
  const
  {
    return std::sqrt(x*x + y*y);
  }
  /// Angle relative to x-axis.
  double angle()
  const
  {
    return std::atan2(y, x);
  }

  /// Add an offset to x and y.
  void offset(double off)
  {
    offset(off, off);
  }
  /// Add an offset to x and an offset to y
  void offset(double  xoff, double yoff)
  {
    x = x + xoff;
    y = y + yoff;
  }

  /// Scale x and y.
  void scale(double mult)
  {
    this->scale(mult, mult);
  }
  /// Scale x and y.
  void scale(double xmult, double ymult)
  {
    this->x = this->x * xmult;
    this->y = this->y * ymult;
  }
  double x;
  double y;
};
```

The scale and offset functions modify data members, so they cannot be const. The angle and distance member functions don't modify any members, so they are const.

Given a point variable, you can call any member function. If the object is const, however, you can call only const member functions. The most common situation is when you find yourself with a const object within another function, and the object was passed by reference to const, as illustrated in Listing 34-4.

Listing 34-4. Calling const and Non-const Member Functions

```
#include <cmath>
#include <iostream>

// Use the same point definition as Listing 34-3
... omitted for brevity ...

void print_polar(point const& pt)
```

```
{
  std::cout << "{ r=" << pt.distance() << ", angle=" << pt.angle() << " }\n";
}

void print_cartesian(point const& pt)
{
  std::cout << "{ x=" << pt.x << ", y=" << pt.y << " }\n";
}

int main()
{
  point p1{}, p2{};
  double const pi{3.141592653589792};
  p1.x = std::cos(pi / 3);
  p1.y = std::sin(pi / 3);
  print_polar(p1);
  print_cartesian(p1);
  p2 = p1;
  p2.scale(4.0);
  print_polar(p2);
  print_cartesian(p2);
  p2.offset(0.0, -2.0);
  print_polar(p2);
  print_cartesian(p2);
}
```

Another common use for member functions is to restrict access to data members. Imagine what would happen if a program that used the BMI `record` type accidentally modified the `bmi_` member. A better design would let you call a `bmi()` function to obtain the BMI but hide the `bmi_` data member, to prevent accidental modification. You can prevent such accidents, and the next Exploration shows you how.

EXPLORATION 35

Access Levels

Everyone has secrets, some of us more than others. Classes have secrets too. For example: Throughout this book, you have used the `std::string` class without having any notion of what goes on inside the class. The implementation details are secrets—not closely guarded secrets, but secrets nonetheless. You cannot directly examine or modify any of `string`'s data members. Instead, it presents quite a few member functions that make up its public interface. You are free to use any of the publicly available member functions, but only the publicly available member functions.
This Exploration explains how you can do the same with your classes.

Public vs. Private

The author of a class determines which members are secrets (for use only by the class's own member functions) and which members are freely available for use by any other bit of code in the program. Secret members are called *private*, and the members that anyone can use are *public*. The privacy setting is called the *access level*. (When you read C++ code, you may see another access level, `protected`. I'll cover that one later. Two access levels are enough to begin with.)

To specify an access level, use the `private` keyword or the `public` keyword, followed by a colon. All subsequent members in the class definition have that accessibility level until you change it with a new access-level keyword. Listing 35-1 shows the `point` class with access-level specifiers.

Listing 35-1. The `point` Class with Access-Level Specifiers

```
struct point
{
public:
  point() : point{0.0, 0.0} {}
  point(double x, double y) : x_{x}, y_{y} {}
  point(point const&) = default;

  double x() const { return x_; }
  double y() const { return y_; }

  double angle()    const { return std::atan2(y(), x()); }
  double distance() const { return std::sqrt(x()*x() + y()*y()); }

  void move_cartesian(double x, double y)
  {
    x_ = x;
    y_ = y;
  }
  void move_polar(double r, double angle)
  {
    move_cartesian(r * std::cos(angle), r * std::sin(angle));
  }

  void scale_cartesian(double s)       { scale_cartesian(s, s); }
  void scale_cartesian(double xs, double ys)
  {
    move_cartesian(x() * xs, y() * ys);
  }
  void scale_polar(double r)           { move_polar(distance() * r, angle()); }
  void rotate(double a)                { move_polar(distance(), angle() + a); }
  void offset(double o)                { offset(o, o); }
  void offset(double xo, double yo)    { move_cartesian(x() + xo, y() + yo); }

private:
  double x_;
  double y_;
};
```

The data members are private, so the only functions that can modify them are `point`'s own member functions. Public member functions provide access to the position with the public `x()` and `y()` member functions.

■ **Tip** Always keep data members private, and provide access only through member functions.

To modify a position, notice that `point` does not let the user arbitrarily assign a new *x* or *y* value. Instead, it offers several public member functions to move the point to an absolute position or relative to the current position.

The public member functions let you work in Cartesian coordinates—that is, the familiar *x* and *y* positions, or in polar coordinates, specifying a position as an angle (relative to the *x* axis) and a distance from the origin. Both representations for a point have their uses, and both can uniquely specify any position in two-dimensional space. Some users prefer polar notation, while others prefer Cartesian. Neither user has direct access to the data members, so it doesn't matter how the `point` class actually stores the coordinates. In fact, you can change the implementation of `point` to store the distance and angle as data members by changing only a few member functions. **Which member functions would you have to change?**

____

____

Changing the data members from `x_` and `y_` to `r_` and `angle_` necessitate a change to the `x`, `y`, `angle`, and `distance` member functions, just for access to the data members. You also have to change the two `move` functions: `move_polar` and `move_cartesian`. Finally, you have to modify the constructors. No other changes are necessary. Because the `scale` and `offset` functions do not access data members directly, but instead call other member functions, they are insulated from changes to the class implementation. **Rewrite the point class to store polar coordinates in its data members**. Compare your class with mine, which is shown in Listing 35-2.

Listing 35-2. The point Class Changed to Store Polar Coordinates

```
struct point
{
public:
  point() : point{0.0, 0.0} {}
  point(double x, double y) : r_{0.0}, angle_{0.0} { move_cartesian(x, y); }
  point(point const&) = default;

  double x() const { return distance() * std::cos(angle()); }
  double y() const { return distance() * std::sin(angle()); }

  double angle()    const { return angle_; }
  double distance() const { return r_; }

  void move_cartesian(double x, double y)
  {
    move_polar(std::sqrt(x*x + y*y), std::atan2(y, x));
  }
  void move_polar(double r, double angle)
  {
    r_ = r;
    angle_ = angle;
  }

  void scale_cartesian(double s)       { scale_cartesian(s, s); }
  void scale_cartesian(double xs, double ys)
  {
    move_cartesian(x() * xs, y() * ys);
  }
  void scale_polar(double r)           { move_polar(distance() * r, angle()); }
  void rotate(double a)                { move_polar(distance(), angle() + a); }
  void offset(double o)                { offset(o, o); }
  void offset(double xo, double yo)    { move_cartesian(x() + xo, y() + yo); }

private:
  double r_;
  double angle_;
};
```

One small difficulty is the constructor. Ideally, point should have two constructors, one taking polar coordinates and the other taking Cartesian coordinates. The problem is that both sets of coordinates are pairs of numbers, and overloading cannot distinguish between the arguments. This means you can't use normal overloading for these constructors. Instead, you can add a third parameter: a flag that indicates whether to interpret the first two parameters as polar coordinates or Cartesian coordinates.

```
polar(double a, double b, bool is_polar)
{
  if (is_polar)
    move_polar(a, b);
  else
    move_cartesian(a, b);
}
```

It's something of a hack, but it will have to do for now. Later in the book, you will learn cleaner techniques to accomplish this task.

class vs. struct

Exploration 34 hinted that the `class` keyword was somehow involved in class definitions, even though every example in this book so far uses the `struct` keyword. Now is the time to learn the truth.

The truth is quite simple. The `struct` and `class` keywords both start class definitions. The only difference is the default access level: `private` for `class` and `public` for `struct`. That's all.

By convention, programmers tend to use `class` for class definitions. A common (but not universal) convention is to start class definitions with the public interface, tucking away the private members at the bottom of the class definition. Listing 35-3 shows the latest incarnation of the `point` class, this time defined using the `class` keyword.

Listing 35-3. The `point` Class Defined with the `class` Keyword

```
class point
{
public:
  point() : r_{0.0}, angle_{0.0} {}

  double x() const { return distance() * std::cos(angle()); }
  double y() const { return distance() * std::sin(angle()); }

  double angle()    const { return angle_; }
  double distance() const { return r_; }

  ... other member functions omitted for brevity ...

private:
  double r_;
  double angle_;
};
```

Plain Old Data

So what good is the `struct` keyword? Authors of introductory books like it, because we can gradually introduce concepts such as classes without miring the reader in too many details, such as access levels, all at once. But what about real-world programs?

The `struct` keyword plays a crucial role in C compatibility. C++ is a distinct language from C, but many programs must interface with the C world. C++ has a couple of key features to interface with C. One of those features is POD. That's right, POD, short for *Plain Old Data*.

The built-in types are POD. A class that has only public POD types as data members, with no constructors and no overloaded assignment operator, is a POD type. A class with a private member, a member with reference or other non-POD type, a constructor, or an assignment operator is not POD.

The importance of POD types is that legacy C functions in the C++ library, in third-party libraries, or operating-system interfaces require POD types. This book won't go into the details of any of these functions, but if you find yourself having to call `memcpy`, `fwrite`, `ReadFileEx`, or any one of the myriad related functions, you will have to make sure you are using POD classes.

POD classes are often declared with struct, and the same header can be used in C and C++ programs. Even if you don't intend to share the header with a C program, using struct in this way emphasizes to the reader that the class is POD. Not everyone uses struct to mean POD, but it's a convention I follow in my own code. I use class for all other cases, to remind the human reader that the class can take advantage of C++ features and is not required to maintain compatibility with C.

Public or Private?

Usually, you can easily determine which members should be public and which should be private. Sometimes, however, you have to stop and ponder. Consider the rational class (last seen in Exploration 33). **Rewrite the rational class to take advantage of access levels**.

Did you decide to make reduce() public or private? I chose private, because there is no need for any outside caller to call reduce(). Instead, the only member functions to call reduce() are the ones that change the data members themselves. Thus, reduce() is hidden from outside view and serves as an implementation detail. The more details you hide, the better, because it makes your class easier to use.

When you added access functions, did you let the caller change the numerator only? Did you write a function to change the denominator only? Or did you ask that the user assign both at the same time? The user of a rational object should treat it as a single entity, a number. You can't assign only a new exponent to a floating-point number, and you shouldn't be able to assign only a new numerator to a rational number. On the other hand, I see no reason not to let the caller examine only the numerator or only the denominator. For example, you may want to write your own output formatting function, which requires knowing the numerator and denominator separately.

A good sign that you have made the right choices is that you can rewrite all the operator functions easily. These functions should not have to access the data members of rational, but use only the public functions. If you tried to access any private members, you learned pretty quickly that the compiler wouldn't let you. That's what privacy is all about.

Compare your solution with my solution, presented in Listing 35-4.

Listing 35-4. The Latest Rewrite of the rational Class

```
#include <cassert>
#include <cstdlib>
#include <iostream>
#include <sstream>

/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
int gcd(int n, int m)
{
  n = std::abs(n);
  while (m != 0) {
    int tmp(n % m);
    n = m;
    m = tmp;
  }
  return n;
}

/// Represent a rational number (fraction) as a numerator and denominator.
class rational
{
public:
  rational(): rational{0}  {}
  rational(int num): numerator_{num}, denominator_{1} {} // no need to reduce
  rational(rational const&) = default;
  rational(int num, int den)
  : numerator_{num}, denominator_{den}
  {
    reduce();
  }

  rational(double r)
  : rational{static_cast<int>(r * 10000), 10000}
  {
    reduce();
  }

  int numerator()   const { return numerator_; }
  int denominator() const { return denominator_; }
  float to_float()
  const
  {
    return static_cast<float>(numerator()) / denominator();
  }

  double to_double()
  const
  {
    return static_cast<double>(numerator()) / denominator();
  }

  long double to_long_double()
  const
  {
    return static_cast<long double>(numerator()) /
           denominator();
  }

  /// Assign a numerator and a denominator, then reduce to normal form.
  void assign(int num, int den)
  {
    numerator_ = num;
    denominator_ = den;
    reduce();
  }
private:
  /// Reduce the numerator and denominator by their GCD.
  void reduce()
  {
    assert(denominator() != 0);
    if (denominator() < 0)
    {
      denominator_ = -denominator();
      numerator_ = -numerator();
    }
    int div{gcd(numerator(), denominator())};
    numerator_ = numerator() / div;
    denominator_ = denominator() / div;
  }

  int numerator_;
  int denominator_;
};

/// Absolute value of a rational number.
rational abs(rational const& r)
{
  return rational{std::abs(r.numerator()), r.denominator()};
}

/// Unary negation of a rational number.
rational operator-(rational const& r)
{
  return rational{-r.numerator(), r.denominator()};
}

/// Add rational numbers.
rational operator+(rational const& lhs, rational const& rhs)
{
  return rational{
          lhs.numerator() * rhs.denominator() + rhs.numerator() * lhs.denominator(),
          lhs.denominator() * rhs.denominator()};
}

/// Subtraction of rational numbers.
rational operator-(rational const& lhs, rational const& rhs)
{
  return rational{
          lhs.numerator() * rhs.denominator() - rhs.numerator() * lhs.denominator(),
          lhs.denominator() * rhs.denominator()};
}

/// Multiplication of rational numbers.
rational operator*(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator()};
}

/// Division of rational numbers.
/// TODO: check for division-by-zero
rational operator/(rational const& lhs, rational const& rhs)
{
  return rational{lhs.numerator() * rhs.denominator(),
                  lhs.denominator() * rhs.numerator()};
}
```

```
/// Compare two rational numbers for equality.
bool operator==(rational const& a, rational const& b)
{
  return a.numerator() == b.numerator() and a.denominator() == b.denominator();
}

/// Compare two rational numbers for inequality.
inline bool operator!=(rational const& a, rational const& b)
{
  return not (a == b);
}
/// Compare two rational numbers for less-than.
bool operator<(rational const& a, rational const& b)
{
  return a.numerator() * b.denominator() < b.numerator() * a.denominator();
}

/// Compare two rational numbers for less-than-or-equal.
inline bool operator<=(rational const& a, rational const& b)
{
  return not (b < a);
}
/// Compare two rational numbers for greater-than.
inline bool operator>(rational const& a, rational const& b)
{
  return b < a;
}

/// Compare two rational numbers for greater-than-or-equal.
inline bool operator>=(rational const& a, rational const& b)
{
  return not (b > a);
}

/// Read a rational number.
/// Format is @em integer @c / @em integer.
std::istream& operator>>(std::istream& in, rational& rat)
{
  int n{}, d{};
  char sep{};
  if (not (in >> n >> sep))
    // Error reading the numerator or the separator character.
    in.setstate(in.failbit);
  else if (sep != '/')
  {
    // Push sep back into the input stream, so the next input operation
    // will read it.
    in.unget();
    rat.assign(n, 1);
  }
  else if (in >> d)
    // Successfully read numerator, separator, and denominator.
    rat.assign(n, d);
  else
    // Error reading denominator.
    in.setstate(in.failbit);

  return in;
}

/// Write a rational numbers.
/// Format is @em numerator @c / @em denominator.
std::ostream& operator<<(std::ostream& out, rational const& rat)
{
  std::ostringstream tmp{};
  tmp << rat.numerator() << '/' << rat.denominator();
  out << tmp.str();

  return out;
}
```

Feel free to save Listing 35-4 as `rational.hpp`, so you can use it and reuse in your own programs. You will revisit this class as you learn more advanced features of C++.

Classes are one of the fundamental building blocks of object-oriented programming. Now that you know how classes work, you can see how they apply to this style of programming, which is the subject of the next Exploration.

EXPLORATION 36

Introduction to Object-Oriented Programming

This Exploration takes a break from C++ programming to turn to the topic of object-oriented programming (OOP). You may already be familiar with this topic, but I urge you to continue reading. You may learn something new. To everyone else, this Exploration introduces some of the foundations of OOP in general terms. Later Explorations will show how C++ implements OOP principles.

Books and Magazines

What is the difference between a book and a magazine? Yes, I really want you to write down your answer. Write down as many differences as you can think of.

____

____

____

What are the similarities between books and magazines? Write down as many similarities as you can think of.

____

____

____

If you can, compare your lists with the lists that other people write. They don't have to be programmers; everyone knows what books and magazines are. Ask your friends and neighbors; stop strangers at the bus stop and ask them. Try to find a core set of commonalities and differences.

Many items on the lists will be qualified. For instance, "most books have at least one author," "many magazines are published monthly," and so on. That's fine. When solving real problems, we often map "maybe" and "sometimes" into "never" or "always," according to the specific needs of the problem at hand. Just remember that this is an OOP exercise, not a bookstore or library exercise.

Now categorize the commonalities and the differences. I'm not telling you how to categorize them. Just try to find a small set of categories that covers the diverse items on your lists. Some less useful categorizations are: group by number of words, group by last letter. **Try to find useful categories. Write them down.**

____

____

____

____

I came up with two broad categories: attributes and actions. *Attributes* describe the physical characteristics of books and magazines.

- Books and magazines have size (number of pages) and cost.
- Most books have an ISBN (International Standard Book Number).
- Most magazines have an ISSN (International Standard Serial Number).
- Magazines have a volume number and issue number.

Books and magazines have a title and publisher. Books have authors. Magazines typically don't. (Magazine articles have authors, but a magazine as a whole rarely lists an author.)

Actions describe how a book or magazine acts or how you interact with them.

- You can read a book or magazine. A book or magazine can be open or closed.
- You can purchase a book or magazine.
- You can subscribe to a magazine.

The key distinction between attributes and actions is that attributes are specific to a single object. Actions are shared by all objects of a common class. Sometimes, actions are called *behaviors*. All dogs exhibit the behavior called panting; they all pant in pretty much the same manner and for the same reasons. All dogs have the attribute color, but one dog is golden, another dog is black, and the dog over there next to the tree is white with black spots.

In programming terms, a *class* describes the behaviors or actions and the types of attributes for all the objects of that class. Each *object* has its own values for the attributes that the class enumerates. In C++ terms, member functions implement actions and provide access to attributes, and data members store attributes.

Classification

Books and magazines don't do much on their own. Instead, their "actions" depend on how we interact with them. A bookstore interacts with books and magazines by selling, stocking, and advertising them. A library's actions include lending and accepting returns. Other kinds of objects have actions they initiate on their own. For example, **what are some of the behaviors of a dog?**

____

____

____

____

____

What are the attributes of a dog?

____

____

____

____

____

What about a cat? **Do cats and dogs have significantly different behaviors?** ____________ **Attributes?** ________________ **Summarize the differences.**

____

____

I don't own dogs or cats, so my observations are limited. From where I sit, dogs and cats have many similar attributes and behaviors. I expect that many readers are much more astute observers than I and can enumerate quite a few differences between the two animals.

Nonetheless, I maintain that once you consider the differences closely, you will see that many of them are not attributes or behaviors unique to one type of animal or the other but are merely different values of a single attribute or different details of a single behavior. Cats may be more fastidious, but dogs and cats both exhibit grooming behavior. Dogs and cats come in different colors, but they both have colored furs (with rare exceptions).

In other words, when trying to enumerate the attributes and behaviors of various objects, your job can be made simpler by classifying similar objects together. For critters, biologists have already done the hard work for us, and they have devised a rich and detailed taxonomy of animals. Thus, a species (*catus* or *familiaris*) belongs to a genus (*Felis* or *Canis*), which is part of a family (Felidae or Canidae). These are grouped yet further into an order (Carnivora), a class (Mammalia), and so on, up to the animal (Metazoa) kingdom. (Taxonomists: Please forgive my oversimplification.)

So what happens to attributes and behaviors as you ascend the taxonomic tree? **Which attributes and behaviors are the same across all mammals?**

____

____

____

____

All animals?

____

____

____

____

As the classification became broader, the attributes and behavior also became more general. Among the attributes of dogs and cats are color of fur, length of tail, weight, and much more. Not all mammals have fur or tails, so you need broader attributes for the entire class. Weight still works, but instead of overall length, you may want to use size. Instead of color of fur, you need only generic coloring. For all animals, the attributes are quite broad: size, weight, single-cell vs. multicell, etc.

Behaviors are similar. You may list that cats purr, dogs pant, both animals can walk and run, and so on. All mammals eat and drink. Female mammals nurse their young. For all animals, you are left with a short, general list: eat and reproduce. It's hard to be more specific than that when you are trying to list the behaviors common to all animals, from amoebae to zebras.

A classification tree helps biologists understand the natural world. Class trees (or *class hierarchies*, as they are often called, because big words make us feel important) help programmers model the natural world in software (or model the unnatural world, as so often happens in many of our projects). Instead of trying to name each level of the tree, programmers prefer a local, recursive view of any class hierarchy. Going up the tree, each class has a *base* class, also called a superclass or parent class. Thus *animal* is a base class of *mammal*, which is a base class of *dog*. Going downward are *derived* classes, also called subclasses or child classes. *Dog* is a derived class of *mammal*. Figure 36-1 illustrates a class hierarchy. Arrows point from derived class to base class.

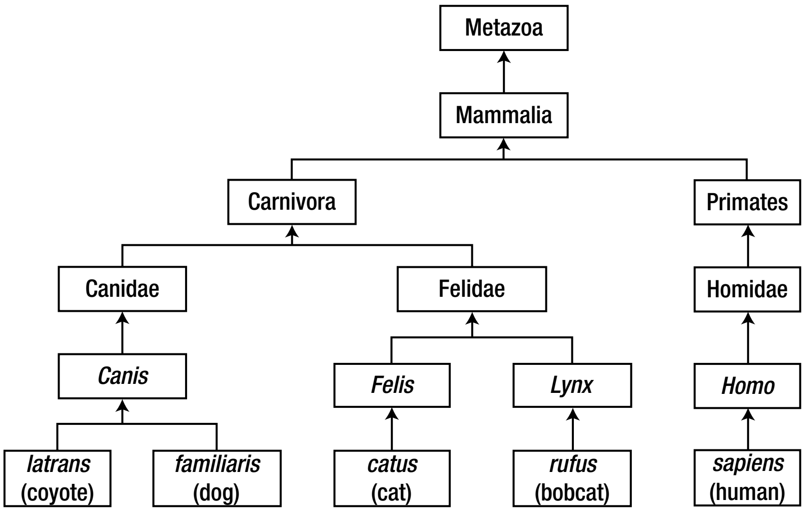


Figure 36-1. *A class diagram*

An *immediate* base class is one with no intervening base classes. For example, the immediate base class of *catus* is *Felis*, which has an immediate base class of Felidae, which has an immediate base class of Carnivora. Metazoa, Mammalia, Carnivora, Felidae, and *Felis* are all base classes of *catus*, but only *Felis* is its immediate base class.

Inheritance

Just as a mammal has all the attributes and behaviors of an animal, and a dog has the attributes and behaviors of all mammals, in an OOP language, a derived class has all the behaviors and attributes of all of its base classes. The term most often used is *inheritance*: the derived class *inherits* the behaviors and attributes of its base class. This term is somewhat unfortunate, because OOP inheritance is nothing like real-world inheritance. When a derived class inherits behaviors, the base class retains its behaviors. In the real world, classes don't inherit anything; objects do.

In the real world, a person object inherits the value of certain attributes (cash, stock, real estate, etc.) from a deceased ancestor object. In the OOP world, a `person` class inherits behaviors from a base class, such as `primate`, by sharing the single copy of those behavior functions that are defined in the base class. A `person` class inherits the attributes of a base class, so objects of the derived class contain values for all the attributes defined in its class and in all of its base classes. In time, the inheritance terminology will become natural to you.

Because inheritance creates a tree structure, tree terminology also pervades discussion of inheritance. As is so common in programming, tree diagrams are drawn upside down, with the root at the top, and leaves at the bottom (as you saw in Figure 36-1). Some OOP languages (Java, Smalltalk, Delphi) have a single root, which is the ultimate base class for all classes. Others, such as C++, do not. Any class can be the root of its own inheritance tree.

So far, the main examples for inheritance involved some form of specialization. *Cat* is more specialized than *mammal*, which is more specialized than *animal*. The same is true in computer programming. For example, class frameworks for graphical user interfaces (GUIs) often use a hierarchy of specialized classes. Figure 36-2 shows a selection of some of the more important classes that make up wxWidgets, which is an open-source C++ framework that supports many platforms.

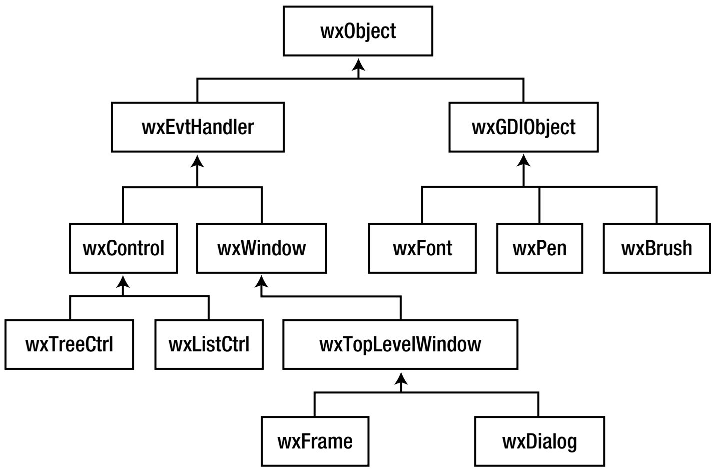


Figure 36-2. *Excerpt from the wxWidgets class hierarchy*

Even though C++ does not require a single root class, some frameworks do; wxWidgets is one that does require a single root class. Most wxWidgets classes derive from `wxObject`. Some objects are straightforward, such as `wxPen` and `wxBrush`. Interactive objects derive from `wxEvtHandler` (short for "event handler"). Thus, each step in the class tree introduces another degree of specialization.

Later in the book, you will see other uses for inheritance, but the most common and most important use is to create specialized derived classes from more general base classes.

Liskov's Substitution Principle

When a derived class specializes the behavior and attributes of a base class (which is the common case), any code that you write involving the base class should work equally well with an object of the derived class. In other words, the act of feeding a mammal is, in broad principles, the same, regardless of the specific kind of animal.

Barbara Liskov and Jeannette Wing formalized this fundamental principle of object-oriented programming, which is often known today as the Substitution Principle or Liskov's Substitution Principle. Briefly, the Substitution Principle states that if you have base class *B* and derived class *D*, in any situation that calls for an object of type *B*, you can substitute an object of type *D,* with no ill effects. In other words, if you need a mammal, any mammal, and someone hands you a dog, you should be able to use that dog. If someone hands you a cat, a horse, or a cow, you can use that animal. If someone hands you a fish, however, you are allowed to reject the fish in any manner that you deem suitable.

The Substitution Principle helps you write programs, but it also imposes a burden. It helps because it frees you to write code that depends on base class behavior without concerning yourself about any derived classes. For example, in a GUI framework, the base `wxEvtHandler` class might be able to recognize a mouse click and dispatch it to an event handler. The click handler does not know or care whether the control is actually a `wxListCtrl` control, a `wxTreeCtrl` control, or a `wxButton`. All that matters is that `wxEvtHandler` accepts a click event, acquires the position, determines which mouse button was clicked, and so on, and then dispatches this event to the event handler.

The burden is on the authors of the `wxButton`, `wxListCtrl`, and `wxTreeCtrl` classes to ensure that their click behavior meets the requirements of the Substitution Principle. The easiest way to meet the requirements is to let the derived class inherit the behavior of the base class. Sometimes, however, the derived class has additional work to do. Instead of inheriting, it provides new behavior. In that case, the programmer must ensure that the behavior is a valid substitution for the base class behavior. The next few Explorations will show concrete examples of this abstract principle.

Type Polymorphism

Before returning to C++-land, I want to present one more general principle. Suppose I hand you a box labeled "Mammal." Inside the box can be any mammal: a dog, a cat, a person, etc. You know the box cannot contain a bird, a fish, a rock, or a tree. It must contain a mammal. Programmers call the box *polymorphic*, from the Greek meaning "many forms." The box can hold any one of many forms, that is, any one mammal, regardless of which form of mammal it is.

Although many programmers use the general term *polymorphism*, this specific kind of polymorphism is *type polymorphism*, also known as *subtyping polymorphism*. That is, the type of a variable (or a box) determines which kinds of objects it can contain. A polymorphic variable (or box) can contain one of a number of types of objects.

In particular, a variable with a base class type can refer to an object of the base class type or to an object of any type that is derived from that base class. According to the Substitution Principle, you can write code to use the base class variable, calling any of the member functions of the base class, and that code will work, regardless of the object's true, derived type.

Now that you have a fundamental understanding of the principles of OOP, it is time to see how they play out in C++.

EXPLORATION 37

Inheritance

The previous Exploration introduced general OOP principles. Now it's time to see how to apply those principles to C++.

Deriving a Class

Defining a derived class is just like defining any other class, except that you include a base class access level and name after a colon. See Listing 37-1 for an example of some simple classes to support a library. Every item in the library is a work of some kind: a book, a magazine, a movie, and so on. To keep things simple, the class `work` has only two derived classes, `book` and `periodical`.

Listing 37-1. Defining a Derived Class

```
class work
{
public:
  work() = default;
  work(work const&) = default;
  work(std::string const& id, std::string const& title) : id_{id}, title_{title} {}
  std::string const& id()    const { return id_; }
  std::string const& title() const { return title_; }
private:
  std::string id_;
  std::string title_;
};

class book : public work
{
public:
  book() : work{}, author_{}, pubyear_{} {}
  book(book const&) = default;
  book(std::string const& id, std::string const& title, std::string const& author,
       int pubyear)
  : work{id, title}, author_{author}, pubyear_{pubyear}
  {}
  std::string const& author() const { return author_; }
  int pubyear()               const { return pubyear_; }
private:
  std::string author_;
  int pubyear_; ///< year of publication
};
```

```
class periodical : public work
{
public:
  periodical() : work{}, volume_{0}, number_{0}, date_{} {}
  periodical(periodical const&) = default;
  periodical(std::string const& id, std::string const& title, int volume,
             int number,
 std::string const& date)
  : work{id, title}, volume_{volume}, number_{number}, date_{date}
  {}
  int volume()              const { return volume_; }
  int number()              const { return number_; }
  std::string const& date() const { return date_; }
private:
  int volume_;       ///< volume number
  int number_;       ///< issue number
  std::string date_; ///< publication date
};
```

When you define a class using the `struct` keyword, the default access level is `public`. For the `class` keyword, the default is `private`. These keywords also affect derived classes. Except in rare circumstances, `public` is the right choice here, which is what I used to write the classes in Listing 37-1.

Also in Listing 37-1, note that there is something new about the initializer lists. A derived class can (and should) initialize its base class by listing the base class name and its initializer. You can call any constructor by passing the right arguments. If you omit the base class from the initializer list, the compiler uses the base class's default constructor.

What do you think happens if the base class does not have a default constructor?

____

Try it. Change `work`'s default constructor from `= default` to `= delete` and try to compile the code for Listing 37-1. (Add a trivial `main()`, to ensure that you write a complete program, and be certain to `#include` all necessary headers.) **What happens?**

____

That's right; the compiler complains. The exact error message or messages you receive vary from compiler to compiler. I get something like the following:

```
$ g++ -ansi -pedantic list3701err.cpp
list3701err.cpp: In constructor 'book::book()':
list3701err.cpp:17:41: error: use of deleted function 'work::work()'
   book() : work{}, author_{}, pubyear_{0} {}
                                         ^
list3701err.cpp:4:3: error: declared here
   work() = delete;
   ^
list3701err.cpp: In constructor 'periodical::periodical()':
list3701err.cpp:33:56: error: use of deleted function 'work::work()'
```

```
   periodical() : work{}, volume_{0}, number_{0}, date_{} {}
                                                     ^
list3701err.cpp:4:3: error: declared here
   work() = delete;
   ^
```

Base classes are always initialized before members, starting with the root of the class tree. You can see this for yourself by writing classes that print messages from their constructors, as demonstrated in Listing 37-2.

Listing 37-2. Printing Messages from Constructors to Illustrate Order of Construction

```
#include <iostream>

class base
{
public:
  base() { std::cout << "base\n"; }
};

class middle : public base
{
public:
  middle() { std::cout << "middle\n"; }
};

class derived : public middle
{
public:
  derived() { std::cout << "derived\n"; }
};

int main()
{
  derived d;
}
```

What output do you expect from the program in Listing 37-2?

____

____

____

Try it. **What output did you actually get?**

____

____

____

Were you correct? ______________ In the interest of being thorough, I receive the following:

```
base
middle
derived
```

Remember that if you omit the base class from the initializers, or you omit the initializer list entirely, the base class's default constructor is called. Listing 37-2 contains only default constructors, so what happens is the constructor for `derived` first invokes the default constructor for `middle`. The constructor for `middle` invokes the default constructor for `base` first, and the constructor for `base` has nothing to do except execute its function body. Then it returns, and the constructor body for `middle` executes and returns, finally letting `derived` run its function body.

Member Functions

A derived class inherits all members of the base class. This means a derived class can call any public member function and access any public data member. So can any users of the derived class. Thus, you can call the `id()` and `title()` functions of a `book` object, and the `work::id()` and `work::title()` functions are called.

The access levels affect derived classes, so a derived class cannot access any private members of a base class. (In Exploration 66, you will learn about a third access level that shields members from outside prying eyes while granting access to derived classes.) Thus, the `periodical` class cannot access the `id_` or `title_` data members, and so a derived class cannot accidentally change a `work`'s identity or title. In this way, access levels ensure the integrity of a class. Only the class that declares a data member can alter it, so it can validate all changes, prevent changes, or otherwise control who changes the value and how.

If a derived class declares a member function with the same name as the base class, the derived class function is the only one visible in the derived class. The function in the derived class is said to *shadow* the function in the base class. As a rule, you want to avoid this situation, but there are several cases in which you very much want to use the same name, without shadowing the base class function. In the next Exploration, you will learn about one such case. Later, you will learn others.

Destructors

When an object is destroyed—perhaps because the function in which it is defined ends and returns—sometimes you have to do some cleanup. A class has another special member function that performs cleanup when an object is destroyed. This special member function is called a *destructor*.

Like constructors, destructors do not have return values. A destructor name is the class name preceded by a tilde (~). Listing 37-3 adds destructors to the example classes from Listing 37-2.

Listing 37-3. Order of Calling Destructors

```
#include <iostream>

class base
{
public:
  base()  { std::cout << "base\n"; }
  ~base() { std::cout << "~base\n"; }
};

class middle : public base
{
public:
  middle()  { std::cout << "middle\n"; }
  ~middle() { std::cout << "~middle\n"; }
};

class derived : public middle
{
public:
  derived()  { std::cout << "derived\n"; }
  ~derived() { std::cout << "~derived\n"; }
};

int main()
{
  derived d;
}
```

What output do you expect from the program in Listing 37-3?

____

____

____

____

____

____

Try it. **What do you actually get?**

____

____

____

____

____

____

Were you correct? ________________ When a function returns, it destroys all local objects in the reverse order of construction. When a destructor runs, it destroys the most-derived class first, by running the destructor's function body. It then invokes the immediate base class destructor. Hence, the destructors run in opposite order of construction in the following example.

```
base
middle
derived
~derived
~middle
~base
```

If you don't write a destructor, the compiler writes a trivial one for you. Whether you write your own, or the compiler implicitly writes the destructor, after every destructor body finishes, the compiler arranges to call the destructor for every data member and then execute the destructor for the base classes, starting with the most-derived. For simple classes in these examples, the compiler's destructors work just fine. Later, you will find more interesting uses for destructors. For now, the main purpose is just to visualize the life cycle of an object.

Read Listing 37-4 carefully.

Listing 37-4. Constructors and Destructors

```
#include <iostream>
#include <vector>

class base
{
public:
  base(int value) : value_{value} { std::cout << "base(" << value << ")\n"; }
  base() : base{0} { std::cout << "base()\n"; }
  base(base const& copy)
  : value_{copy.value_}
  { std::cout << "copy base(" << value_ << ")\n"; }

  ~base() { std::cout << "~base(" << value_ << ")\n"; }
  int value() const { return value_; }
  base& operator++()
  {
    ++value_;
    return *this;
  }
private:
  int value_;
};

class derived : public base
{
public:
  derived(int value): base{value} { std::cout << "derived(" << value << ")\n"; }
  derived() : base{} { std::cout << "derived()\n"; }
  derived(derived const& copy)
  : base{copy}
  { std::cout << "copy derived(" << value() << "\n"; }
  ~derived() { std::cout << "~derived(" << value() << ")\n"; }
};

derived make_derived()
{
  return derived{42};
}

base increment(base b)
{
  ++b;
  return b;
}

void increment_reference(base& b)
{
  ++b;
}

int main()
{
  derived d{make_derived()};
  base b{increment(d)};
  increment_reference(d);
  increment_reference(b);
  derived a(d.value() + b.value());
}
```

Fill in the left-hand column of Table 37-1 with the output you expect from the program.

Table 37-1. *Expected and Actual Results of Running the Program in Listing 37-4*

Expected Output	Actual Output

Try it. Fill in the right-hand column of Table 37-1 with the actual output and compare the two columns. **Did you get everything correct?** ________________.

Below is the output generated on my system, along with some commentary. Remember that compilers have some leeway in optimizing away extra calls to the copy constructor. You may get one or two extra copy calls in the mix.

```
base(42)                       // inside make_derived()
derived(42)                    // finish constructing in make_derived()
copy base(42)                  // copy to b in call to increment()
copy base(43)                  // copy return value from increment to b in main
~base(43)                      // destroy temporary return value
base(87)                       // construct a in main
derived(87)                    // construct a in main
~derived(87)                   // end of main: destroy a
~base(87)                      // destroy a
~base(44)                      // destroy b
~derived(43)                   // destroy d
~base(43)                      // finish destroying d
```

Note how pass-by-reference (increment_reference) does not invoke any constructors, because no objects are being constructed. Instead, references are passed to the function, and the referenced object is incremented.

By the way, I have not yet shown you how to overload the increment operator, but you probably guessed that's how it works (in class base). Decrement is similar.

Access Level

At the start of this Exploration, I advised you to use public before the base class name but never explained why. Now is the time to fill you in on the details.

Access levels affect inheritance the same way they affect members. *Public inheritance* occurs when you use the struct keyword to define a class or use the public keyword before the base class name. Public inheritance means the derived class inherits every member of the base class at the same access level that the members have in the base class. Except in rare circumstances, this is exactly what you want.

Private inheritance occurs when you use the private keyword, and it is the default when you define a class using the class keyword. Private inheritance keeps every member of the base class private and inaccessible to users of the derived class. The compiler still calls the base class constructor and destructor when necessary, and the derived class still inherits all the members of the base class. The derived class can call any of the base class's public member functions, but no one else can call them through the derived class. It's as though the derived class re-declares all inherited members as private. Private inheritance lets a derived class make use of the base class without being required to meet the Substitution Principle. This is an advanced technique, and I recommend that you try it only with proper adult supervision.

If the compiler complains about inaccessible members, most likely you forgot to include a public keyword in the class definition. Try compiling Listing 37-5 to see what I mean.

Listing 37-5. Accidentally Inheriting Privately

```
class base
{
public:
  base(int v) : value_{v} {}
  int value() const { return value_; }
private:
  int value_;
};
```

```
class derived : base
{
public:
  derived() : base{42} {}
};

int main()
{
  base b{42};
  int x{b.value()};
  derived d{};
  int y{d.value()};
}
```

The compiler issues an error message, complaining that `base` is private or not accessible from `derived`, or something along those lines.

Programming Style

When in doubt, make data members and member functions private, unless and until you know you need to make a member public. Once a member is part of the public interface, anyone using your class is free to use that member, and you have one more code dependency. Changing a public member means finding and fixing all those dependencies. Keep the public interface as small as possible. If you have to add members later, you can, but it's much harder to remove a member or change it from public to private. Anytime you have to add members to support the public interface, make the supporting functions and data members private.

Use public, not private, inheritance. Remember that inherited members also become part of the derived class's public interface. If you change which class is the base class, you may have to write additional members in the derived class, to make up for members that were in the original base class but are missing from the new base class. The next Exploration continues the discussion of how derived classes work with base classes to provide important functionality.

EXPLORATION 38

Virtual Functions

Deriving classes is fun, but there's not a lot you can do with them—at least, not yet. The next step is to see how C++ implements type polymorphism, and this Exploration starts you on that journey.

Type Polymorphism

Recall from Exploration 36 that type polymorphism is the ability of a variable of type B to take the "form" of any class derived from B. The obvious question is: "How?" The key in C++ is to use a magic keyword to declare a member function in a base class and also implement the function in a derived class with a different magic word. The magic keyword tells the compiler that you want to invoke type polymorphism, and the compiler implements the polymorphism magic. Define a variable of type reference-to-base class and initialize it with an object of derived class type. When you call the polymorphic function for the object, the compiled code checks the object's actual type and calls the derived class implementation of the function. The magic word to turn a function into a polymorphic function is `virtual`. Derived classes are marked with `override`.

For example, suppose you want to be able to print any kind of work in the library (see Listing 37-1) using standard (more or less) bibliographical format. For books, I use the format

author, title, year.

For periodicals, I use

title, volume(number), date.

Add a `print` member function to each class, to print this information. Because this function has different behavior in each derived class, the function is polymorphic, so use the `virtual` keyword before the base class declaration of `print` and `override` after each derived class declaration, as shown in Listing 38-1.

Listing 38-1. Adding a Polymorphic `print` Function to Every Class Derived from `work`

```
class work
{
public:
  work() = default;
  work(work const&) = default;
  work(std::string const& id, std::string const& title) : id_{id}, title_{title} {}
  virtual ~work() {}
  std::string const& id()    const { return id_; }
  std::string const& title() const { return title_; }
  virtual void print(std::ostream&) const {}
private:
  std::string id_;
  std::string title_;
};

class book : public work
{
public:
  book() : work{}, author_{}, pubyear_{0} {}
  book(book const&) = default;
  book(std::string const& id, std::string const& title, std::string const& author,
       int pubyear)
  : work{id, title}, author_{author}, pubyear_{pubyear}
  {}
  std::string const& author() const { return author_; }
  int pubyear()               const { return pubyear_; }
  void print(std::ostream& out)
  const override
  {
    out << author() << ", " << title() << ", " << pubyear() << ".";
  }
private:
  std::string author_;
  int pubyear_; ///< year of publication
};

class periodical : public work
{
public:
  periodical() : work{}, volume_{0}, number_{0}, date_{} {}
  periodical(periodical const&) = default;
  periodical(std::string const& id, std::string const& title, int volume,
             int number,
 std::string const& date)
  : work{id, title}, volume_{volume}, number_{number}, date_{date}
  {}
  int volume()              const { return volume_; }
  int number()              const { return number_; }
  std::string const& date() const { return date_; }
  void print(std::ostream& out)
  const override
  {
    out << title() << ", " << volume() << '(' << number() << "), " <<
           date() << ".";
  }
private:
  int volume_;       ///< volume number
  int number_;       ///< issue number
  std::string date_; ///< publication date
};
```

■ **Tip** When writing a stub function, such as `print()`, in the `base` class, omit the parameter name or names. The compiler requires only the parameter types. Some compilers warn you if a parameter or variable is not used, and even if the compiler doesn't issue a warning, it is a clear message to the human who reads your code that the parameters are not used.

A program that has a reference to a `work` object can call the `print` member function to print that work, and because `print` is polymorphic, or virtual, the C++ environment performs its magic to ensure that the correct `print` is called, depending on whether the `work` object is actually a `book` or a `periodical`. To see this demonstrated, read the program in Listing 38-2.

Listing 38-2. Calling the `print` Function

```
#include <iostream>
#include <string>

// All of Listing 38-1 belongs here
... omitted for brevity ...

void showoff(work const& w)
{
  w.print(std::cout);
  std::cout << '\n';
}

int main()
{
  book sc{"1", "The Sun Also Crashes", "Ernest Lemmingway", 2000};
  book ecpp{"2", "Exploring C++", "Ray Lischner", 2013};
  periodical pop{"3", "Popular C++", 13, 42, "January 1, 2000"};
  periodical today{"4", "C++ Today", 1, 1, "January 13, 1984"};

  showoff(sc);
  showoff(ecpp);
  showoff(pop);
  showoff(today);
}
```

What output do you expect?

____

____

____

____

Try it. **What output do you actually get?**

____

____

____

____

The showoff function does not have to know about the book or periodical classes. As far as it is concerned, w is a reference to a work object. The only member functions you can call are those declared in the work class. Nonetheless, when showoff calls print, it will invoke book's print or periodical's print, if the object's true type is book or periodical.

Write an output operator (`operator<<`) that prints a `work` object by calling its `print` member function. Compare your solution with my solution, as shown in Listing 38-3.

Listing 38-3. Output Operator for Class work

```
std::ostream& operator<<(std::ostream& out, work const& w)
{
  w.print(out);
  return out;
}
```

Writing the output operator is perfectly normal. Just be certain you declare w as a reference. Polymorphic magic does not occur with ordinary objects, only references. With this operator, you can write any work-derived object to an output stream, and it will print using its print function.

■ **Tip** The const keyword, if present, always comes before override. Although the specifiers, such as virtual, can be mixed freely with the function's return type (even if the result is strange, such as int virtual long function()), the const qualifier and override specifier must follow a strict order.

Virtual Functions

A polymorphic function is called a *virtual function* in C++, owing to the virtual keyword. Once a function is defined as virtual, it remains so in every derived class. The virtual function must have the same name, the same return type, and the same number and type of parameters (but the parameters can have different names) in the derived class.

A derived class is not required to implement a virtual function. If it doesn't, it inherits the base class function the same way it does for a non-virtual function. When a derived class implements a virtual function, it is said to *override* the function, because the derived class's behavior overrides the behavior that would have been inherited from the base class.

In the derived class, the override specifier is optional but helps to prevent mistakes. If you accidentally mistype the function's name or parameters in the derived class, the compiler might think you are defining a brand-new function. By adding override, you tell the compiler that you intend to override a virtual function that was declared in the base class. If the compiler cannot find a matching function in the base class, it issues an error message.

Add a class, `movie`, to the library classes. The movie class represents a movie or film recording on tape or disc. Like book and periodical, the movie class derives from work. For the sake of simplicity, define a movie as having an integer running time (in minutes), in addition to the members it inherits from work. Do not override print yet. Compare your class to Listing 38-4.

Listing 38-4. Adding a Class `movie`

```
class movie : public work
{
public:
  movie() : work{}, runtime_{0} {}
  movie(movie const&) = default;
  movie(std::string const& id, std::string const& title, int runtime)
  : work{id, title}, runtime_{runtime}
  {}
  int runtime() const { return runtime_; }
private:
  int runtime_; ///< running length in minutes
};
```

Now modify the test program from Listing 38-2 to create and print a `movie` object. If you want, you can take advantage of the new output operator, instead of calling `showoff`. Compare your program with Listing 38-5.

Listing 38-5. Using the New `movie` Class

```
#include <iostream>
#include <string>

// All of Listing 38-1 belongs here
// All of Listing 38-3 belongs here
// All of Listing 38-4 belongs here
... omitted for brevity ...

int main()
{
  book sc{"1", "The Sun Also Crashes", "Ernest Lemmingway", 2000};
  book ecpp{"2", "Exploring C++", "Ray Lischner", 2006};
  periodical pop{"3", "Popular C++", 13, 42, "January 1, 2000"};
  periodical today{"4", "C++ Today", 1, 1, "January 13, 1984"};
  movie tr{"5", "Lord of the Token Rings", 314};

  std::cout << sc << '\n';
  std::cout << ecpp << '\n';
  std::cout << pop << '\n';
  std::cout << today << '\n';
  std::cout << tr << '\n';
}
```

What do you expect as the last line of output?

____

Try it. **What do you get?**

____

Because `movie` does not override `print`, it inherits the implementation from the base class, `work`. The definition of `print` in the `work` class does nothing, so printing the `tr` object prints nothing.

Fix the problem by adding `print` to the movie class. Now your `movie` class should look something like Listing 38-6.

Listing 38-6. Adding a `print` Member Function to the `movie` Class

```
class movie : public work
{
public:
  movie() : work{}, runtime_{0} {}
  movie(movie const&) = default;
  movie(std::string const& id, std::string const& title, int runtime)
  : work{id, title}, runtime_{runtime}
  {}
  int runtime() const { return runtime_; }
  void print(std::ostream& out)
  const override
  {
    out << title() << " (" << runtime() << " min)";
  }
private:
  int runtime_; ///< running length in minutes
};
```

The `override` keyword is optional in the derived class but highly recommended. Some programmers also use the `virtual` keyword in the derived classes. In C++ 03, this served as a reminder to the human reader that the derived class function overrides a virtual function. The `override` specifier was added in C++ 11 and has the added feature of telling the compiler the same thing, so the compiler can check your work and complain if you make a mistake. I urge you to use `override` everywhere it belongs.

EVOLUTION OF A LANGUAGE

You may find it odd that the `virtual` keyword appears at the start of a function header and `override` appears at the end. You are witnessing the compromises that are often necessary when a language evolves.

The `override` specifier is new in C++ 11. One way to add the `override` specifier would have been to add it to the list of function specifiers, like `virtual`. But adding a new keyword to a language is fraught with difficulty. Every existing program that uses `override` as a variable or other user-defined name would break. Programmers all over the world would have to check and possibly modify their software to avoid this new keyword.

So the C++ standard committee devised a way to add `override` without making it a reserved keyword. The syntax of a function declaration puts the `const` qualifier in a special place. No other identifiers are allowed there, so it is easy to add `override` to the syntax for member functions in a manner similar to `const`, and with no risk of breaking existing code.

Other new language features use existing keywords in new ways, such as `=default` and `=delete` for constructors. But a few new keywords were added, and they bring with them the risk of breaking existing code. So the committee tried to choose names that would be less likely to conflict with existing user-chosen names. You will see examples of some of these new keywords later in the book.

References and Slices

The showoff function in Listing 38-2 and the output operator in Listing 38-3 declare their parameter as a reference to const work. **What do you expect would happen if you were to change them to pass-by-value?**

____

____

____

Try it. Delete the ampersand in the declaration of the output operator, as shown in the following:

```
std::ostream& operator<<(std::ostream& out, work w)
{
  w.print(out);
  return out;
}
```

Run the test program from Listing 38-5. **What is the actual output?**

____

____

____

____

____

Explain what happened.

____

____

____

When you pass an argument by value or assign a derived class object to a base class variable, you lose polymorphism. For instance, instead of a book, the result is an honest-to-goodness, genuine, no-artificial-ingredients work—with no memory of book-ness whatsoever. Thus, the output operator ends up calling work's version of print every time the output operator calls it. That's why the program's output is a bunch of empty lines. When you pass a book object to the output operator, not only do you lose polymorphism, but you also lose all sense of book-ness. In particular, you lose the author_ and pubyear_ data members. The data members that a derived class adds are *sliced* away when the object is copied to a base class variable. Another way to look at it is this: because the derived class members are sliced away, what is left is only a work object, so you cannot have polymorphism. The same thing occurs with assignment.

```
work w;
book nuts{"7", "C++ in a Nutshell", "Ray Lischner", 2003};
w = nuts; // slices away the author_ and pubyear_; copies only id_ and title_
```

Slicing is easy to avoid when writing functions (pass all arguments by reference) but harder to cope with for assignment. The techniques you need to manage assignment come much later in this book. For now, I will focus on writing polymorphic functions.

Pure Virtual Functions

The class work defines the print function, but the function doesn't do anything useful. In order to be useful, every derived class must override print. The author of a base class, such as work, can ensure that every derived class properly overrides a virtual function, by omitting the body of the function and substituting the tokens, = 0, instead. These tokens mark the function as a *pure virtual function*, which means the function has no implementation to inherit, and derived classes must override the function.

Modify the work class to make print a pure virtual function. Then delete the book class's print function, just to see what happens. What does happen?

____

____

The compiler enforces the rules for pure virtual functions. A class that has at least one pure virtual function is said to be *abstract*. You cannot define an object of abstract type. **Fix the program**. The new work class should look something like Listing 38-7.

Listing 38-7. Defining work As an Abstract Class

```
class work
{
public:
  work() = default;
  work(work const&) = default;
  work(std::string const& id, std::string const& title) : id_(id), title_(title) {}
  virtual ~work() {}
  std::string const& id()    const { return id_; }
  std::string const& title() const { return title_; }
  virtual void print(std::ostream& out) const = 0;
private:
  std::string id_;
  std::string title_;
};
```

Virtual Destructors

Although most classes you are writing at this time do not require destructors, I want to mention an important implementation rule. Any class that has virtual functions must declare its destructor to be virtual too. This rule is a programming guideline, not a semantic requirement, so the compiler will not help you by issuing a message when you break it (although some compilers may issue a warning). Instead, you must enforce this rule yourself, through discipline.

I will repeat the rule when you begin to write classes that require destructors. If you try any experiments on your own, please be mindful of this rule, or else your programs could be subject to subtle problems—or not-so-subtle crashes.

The next Exploration continues the discussion of classes and their relationship in the C++ type system.

EXPLORATION 39

Classes and Types

One of the main design goals for C++ was to give the programmer the ability to define custom types that look and act nearly identically to the built-in types. The combination of classes and overloaded operators gives you that power. This Exploration takes a closer look at the type system and how your classes can best fit into the C++ world.

Classes vs. typedefs

Suppose you are writing a function to compute body-mass index (BMI) from an integer height in centimeters and an integer weight in kilograms. You have no difficulty writing such a function (which you can copy from your work in Explorations 28 and 34). For added clarity, you decide to add `typedef`s for `height` and `weight`, which allows the programmer to define variables for storing and manipulating these values with extra clarity to the human reader. Listing 39-1 shows a simple use of the `compute_bmi()` function and the associated `typedef`s.

Listing 39-1. Computing BMI

```
#include <iostream>

typedef int height;
typedef int weight;
typedef int bmi;

bmi compute_bmi(height h, weight w)
{
  return w * 10000 / (h * h);
}

int main()
{
  std::cout << "Height in centimeters: ";
  height h{};
  std::cin >> h;

  std::cout << "Weight in kilograms: ";
  weight w{};
  std::cin >> w;

  std::cout << "Body-mass index = " << compute_bmi(w, h) << '\n';
}
```

Test the program. **What's wrong?**

____

____

If you haven't spotted it yet, take a closer look at the call to compute_bmi, on the last line of code in main(). Compare the arguments with the parameters in the function definition. Now do you see the problem?

In spite of the extra clarity that the height and weight typedefs offer, I still made a fundamental mistake and reversed the order of the arguments. In this case, the error is easy to spot, because the program is small. Also, the program's output is so obviously wrong that testing quickly reveals the problem. Don't relax too much, though; not all mistakes are so obvious.

The problem here is that a typedef does not define a new type but instead creates an alias for an existing type. The original type and its typedef alias are completely interchangeable. Thus, a height is the same as an int is the same as a weight. Because the programmer is able to mix up height and weight, the typedefs don't actually help much.

More useful would be to create distinct types called height and weight. As distinct types, you would not be able to mix them up, and you would have full control over the operations that you allow. For example, dividing two weights should yield a plain, unitless int. Adding a height to a weight should result in an error message from the compiler. Listing 39-2 shows simple height and weight classes that impose these restrictions.

Listing 39-2 Defining Classes for height and weight

```
#include <iostream>

/// Height in centimeters
class height
{
public:
  height(int h) : value_{h} {}
  int value() const { return value_; }
private:
  int value_;
};

/// Weight in kilograms
class weight
{
public:
  weight(int w) : value_{w} {}
  int value() const { return value_; }
private:
  int value_;
};

/// Body-mass index
class bmi
{
public:
  bmi() : value_{0} {}
  bmi(height h, weight w) : value_{w.value() * 10000 / (h.value()*h.value())} {}
  int value() const { return value_; }
private:
  int value_;
};
```

```
height operator+(height a, height b)
{
  return height{a.value() + b.value()};
}
int operator/(height a, height b)
{
  return a.value() / b.value();
}
std::istream& operator>>(std::istream& in, height& h)
{
  int tmp{};
  if (in >> tmp)
    h = tmp;
  return in;
}
std::istream& operator>>(std::istream& in, weight& w)
{
  int tmp{};
  if (in >> tmp)
    w = tmp;
  return in;
}
std::ostream& operator<<(std::ostream& out, bmi i)
{
  out << i.value();
  return out;
}
// Implement other operators similarly, but implement only
// the ones that make sense.
weight operator-(weight a, weight b)
{
  return weight{a.value() - b.value()};
}
... plus other operators you want to implement ...


int main()
{
  std::cout << "Height in centimeters: ";
  height h{0};
  std::cin >> h;

  std::cout << "Weight in kilograms: ";
  weight w{0};
  std::cin >> w;

  std::cout << "Body-mass index = " << bmi(h, w) << '\n';
}
```

The new classes prevent mistakes, such as that in Listing 39-1, but at the expense of more code. For example, you have to write suitable I/O operators. You also have to decide which arithmetic operators to implement. And don't forget the comparison operators. Most of these functions are trivial to write, but you can't neglect them. In many applications, however, the work will pay off many times over, by removing potential sources of error.

I'm not suggesting that you do away with unadorned integers and other built-in types and replace them with clumsy wrapper classes. In fact, I agree with you (don't ask how I know what you're thinking) that the BMI example is rather artificial. If I were writing a real, honest-to-goodness program for computing and managing BMIs, I would use plain `int` variables and rely on careful coding and proofreading to prevent and detect errors. I use wrapper classes, such as `height` and `weight`, when they add some primary value. A big program in which heights and weights figured prominently would offer many opportunities for mistakes. In that case, I would want to use wrapper classes. I could also add some error checking to the classes, impose constraints on the domain of values they can represent, or otherwise help myself to do my job as a programmer. Nonetheless, it's best to start simple and add complexity slowly and carefully. The next section explains in greater detail what behavior you must implement to make a useful and meaningful custom class.

Value Types

The `height` and `weight` types are examples of *value types*—that is, types that behave as ordinary values. Contrast them with the I/O stream types, which behave very differently. For example, you cannot copy or assign streams; you must pass them by reference to functions. Nor can you compare streams or perform arithmetic on them. Value types, by design, behave similarly to the built-in types, such as `int` and `float`. One of the important characteristics of value types is that you can store them in containers, such as `vector` and `map`. This section explains the general requirements for value types.

The basic guideline is to make sure your type behaves "like an `int`." When it comes to copying, comparing, and performing arithmetic, avoid surprises, by making your custom type look, act, and work as much like the built-in types as possible.

Copying

Copying an `int` yields a new `int` that is indistinguishable from the original. Your custom type should behave the same way.

Consider the example of `string`. Many implementations of `string` are possible. Some of these use copy-on-write to optimize frequent copying and assignment. In a copy-on-write implementation, the actual string contents are kept separate from the `string` object. Copies of the `string` object do not copy the contents until and unless a copy is needed, which happens when the string contents must be modified. Many uses of strings are read-only, so copy-on-write avoids unnecessary copies of the contents, even when the `string` objects themselves are copied frequently.

Other implementations optimize for small strings by using the `string` object to store their contents but storing large strings separately. Copying small strings is fast, but copying large strings is slower. Most programs use only small strings. In spite of these differences in implementation, when you copy a `string` (such as passing a `string` by value to a function), the copy and the original are indistinguishable, just like an `int`.

Usually, the automatic copy constructor does what you want, and you don't have to write any code. Nonetheless, you have to think about copying and assure yourself that the compiler's automatic (also called *implicit*) copy constructor does exactly what you want.

Assigning

Assigning objects is similar to copying them. After an assignment, the target and source must contain identical values. The key difference between assignment and copying is that copying starts with a blank slate: a newly constructed object. Assignment begins with an existing object, and you may have to clean up the old value before you can assign the new value. Simple types such as `height` have nothing to clean up, but later in this book, you will learn how to implement more complicated types, such as `string`, which require careful cleanup.

Most simple types work just fine with the compiler's implicit assignment operator, and you don't have to write your own. Nonetheless, you must consider the possibility and make sure the implicit assignment operator is exactly what you want.

Moving

Sometimes, you don't want to make an exact copy. I know I wrote that assignment should make an exact copy, but you can break that rule by having assignment *move* a value from the source to the target. The result leaves the source in an unknown state (typically empty), and the target gets the original value of the source.

Force a move assignment by calling `std::move` (declared in `<utility>`):

```
std::string source{"string"}, target;
target = std::move(source);
```

After the assignment, `source` is in an unknown, but valid, state. Typically it will be empty, but you cannot write code that assumes it is empty. In practical terms, the string contents of `source` are moved into target without copying any of the string contents. Moving is fast and independent of the amount of data stored in a container.

You can also move an object in an initializer, as follows:

```
std::string source{"string"};
std::string target{std::move(source)};
```

Moving works with strings and most containers. Consider the program in Listing 39-3.

Listing 39-3. Copying vs. Moving

```
#include <iostream>
#include <utility>
#include <vector>

void print(std::vector<int> const& vector)
{
  std::cout << "{ ";
  for (int i : vector)
    std::cout << i << ' ';
  std::cout << "}\n";
}

int main()
{
  std::vector<int> source{1, 2, 3 };
  print(source);
  std::vector<int> copy{source};
  print(copy);
  std::vector<int> move{std::move(source)};
  print(move);
  print(source);
}
```

Predict the output of the program in Listing 39-3.

____

____

____

____

When I run the program, I get the following:

```
{ 1 2 3 }
{ 1 2 3 }
{ 1 2 3 }
{ }
```

The first three lines print `{ 1 2 3 }`, as expected. But the last line is interesting, because `source` was moved into `move`. When a vector's contents are moved, the source becomes empty. Writing a move constructor is advanced and will have to wait until later in this book, but you can take advantage of move constructors and move assignment operators in the standard library by calling `std::move()`.

Comparing

I defined copying and assignment in a way that requires meaningful comparison. If you can't determine whether two objects are equal, you can't verify whether you copied or assigned them correctly. C++ has a few ways to check whether two objects are the same.

- The first and most obvious way is to compare objects with the == operator. Value types should overload this operator. Make sure the operator is transitive—that is, if `a == b` and `b == c`, then `a == c`. Make sure the operator is commutative, that is, if `a == b`, then `b == a`. Finally, the operator should be reflexive: `a == a`.
- Standard algorithms such as `find` compare items by one of two methods: with `operator==` or with a caller-supplied predicate. Sometimes, you may want to compare objects with a custom predicate, for example, a `person` class might have `operator==` that compares every data member (name, address, etc.), but you want to search a container of `person` objects by checking only last names, which you do by writing your own comparison function. The custom predicate must obey the same transitive and reflexive restrictions as the == operator. If you are using the predicate with a specific algorithm, that algorithm calls the predicate in a particular way, so you know the order of the arguments. You don't have to make your predicate commutative, and in some cases, you wouldn't want to.
- Containers such as `map` store their elements in sorted order. Some standard algorithms, such as `binary_search`, require their input range to be in sorted order. The ordered containers and algorithms use the same conventions. By default, they use the < operator, but you can also supply your own comparison predicate. These containers and algorithms never use the == operator to determine whether two objects are the same. Instead, they check for equivalence—that is, `a` is equivalent to `b` if `a < b` is false and `b < a` is false.

  If your value type can be ordered, you should overload the < operator. Ensure that the operator is transitive (if `a < b` and `b < c`, then `a < c`). Also, the ordering must be strict, that is, `a < a` is always false.

- Containers and algorithms that check for equivalence also take an optional custom predicate instead of the < operator. The custom predicate must obey the same transitive and strictness restrictions as the < operator.

Not all types are comparable with a less-than relationship. If your type cannot be ordered, do not implement the < operator, but you must also understand that you will not be able to store objects of that type in a `map` or use any of the binary search algorithms. Sometimes, you may want to impose an artificial order, just to permit these uses. For example, a `color` type may represent colors such as `red`, `green`, or `yellow`. Although nothing about `red` or `green` inherently defines one as being "less than" another, you may want to define an arbitrary order, just so you can use these values as keys in a `map`. One immediate suggestion is to write a comparison function that compares colors as integers, using the < operator.

On the other hand, if you have a value that should be compared (such as `rational`), you should implement `operator==` and `operator<`. You can then implement all other comparison operators in terms of these two. (See Exploration 32 for an example of how the `rational` class does this.)

(If you have to store unordered objects in a `map`, you can use `std::unordered_map`, which is a new type in C++ 11. It works almost exactly the same as `std::map`, but it stores values in a hash table instead of a binary tree. Ensuring that a custom type can be stored in `std::unordered_map` is more advanced and won't be covered until much later.)

Implement a color class that describes a color as three components: red, green, and blue, which are integers in the range 0 to 255. Define a comparison function, `order_color`, to permit storing colors as `map` keys. **For extra credit, devise a suitable I/O format and overload the I/O operators too.** Don't worry about error-handling yet—for example, what if the user tries to set red to 1000, blue to 2000, and green to 3000. You'll get to that soon enough.

Compare your solution with mine, which is presented in Listing 39-4.

Listing 39-4. The `color` Class

```
#include <iomanip>
#include <iostream>
#include <sstream>

class color
{
public:
  color() : color{0, 0, 0} {}
  color(color const&) = default;
  color(int r, int g, int b) : red_{r}, green_{g}, blue_{b} {}
  int red() const { return red_; }
  int green() const { return green_; }
  int blue() const { return blue_; }
  /// Because red(), green(), and blue() are supposed to be in the range [0,255],
  /// it should be possible to add them together in a single long integer.
  /// TODO: handle out of range
  long int combined() const { return ((red() * 256L + green()) * 256) + blue(); }
private:
  int red_, green_, blue_;
};

inline bool operator==(color const& a, color const& b)
{
  return a.combined() == b.combined();
}
```

```
inline bool operator!=(color const& a, color const& b)
{
  return not (a == b);
}

inline bool order_color(color const& a, color const& b)
{
  return a.combined() < b.combined();
}

/// Write a color in HTML format: #RRGGBB.
std::ostream& operator<<(std::ostream& out, color const& c)
{
  std::ostringstream tmp{};
  // The hex manipulator tells a stream to write or read in hexadecimal (base 16).
  tmp << '#' << std::hex << std::setw(6) << std::setfill('0') << c.combined();
  out << tmp.str();
  return out;
}

class ioflags
{
public:
  /// Save the formatting flags from @p stream.
  ioflags(std::basic_ios<char>& stream) : stream_{stream}, flags_{stream.flags()} {}
  ioflags(ioflags const&) = delete;
  /// Restore the formatting flags.
  ~ioflags() { stream_.flags(flags_); }
private:
  std::basic_ios<char>& stream_;
  std::ios_base::fmtflags flags_;
};

std::istream& operator>>(std::istream& in, color& c)
{
  ioflags flags{in};

  char hash{};
  if (not (in >> hash))
    return in;
  if (hash != '#')
  {
    // malformed color: no leading # character
    in.unget();                 // return the character to the input stream
    in.setstate(in.failbit);    // set the failure state
    return in;
  }
  // Read the color number, which is hexadecimal: RRGGBB.
  int combined{};
  in >> std::hex >> std::noskipws;
  if (not (in >> combined))
    return in;
  // Extract the R, G, and B bytes.
  int red, green, blue;
  blue = combined % 256;
  combined = combined / 256;
  green = combined % 256;
  combined = combined / 256;
  red = combined % 256;

  // Assign to c only after successfully reading all the color components.
  c = color{red, green, blue};

  return in;
}
```

Listing 39-3 introduced a new trick with the `ioflags` class. The next section explains all.

Resource Acquisition Is Initialization

A programming idiom that goes by the name of Resource Acquisition Is Initialization (RAII) takes advantage of constructors, destructors, and automatic destruction of objects when a function returns. Briefly, the RAII idiom means that a constructor acquires a resource: it opens a file, connects to a network, or even just copies some flags from an I/O stream. The acquisition is part of the object's initialization. The destructor releases the resource: closes the file, disconnects from the network, or restores any modified flags in the I/O stream.

To use an RAII class, all you have to do is define an object of that type. That's all. The compiler takes care of the rest. The RAII class's constructor takes whatever arguments it needs to acquire its resources. When the surrounding function returns, the RAII object is automatically destroyed, thereby releasing the resources. It's that simple.

You don't even have to wait until the function returns. Define an RAII object in a compound statement, and the object is destroyed when the statement finishes and control leaves the compound statement.

- The `ioflags` class in Listing 39-4 is an example of using RAII. It throws some new items at you; let's take them one at a time.
- The `std::basic_ios<char>` class is the base class for all I/O stream classes, such as `istream` and `ostream`. Thus, `ioflags` works the same with input and output streams.
- The `std::ios_base::fmtflags` type is the type for all the formatting flags.
- The `flags()` member function with no arguments returns all the current formatting flags.
- The `flags()` member function with one argument sets all the flags to its argument.

The way to use `ioflags` is simply to define a variable of type `ioflags` in a function or compound statement, passing a stream object as the sole argument to the constructor. The function can change any of the stream's flags. In this case, the input operator sets the input radix (or base) to hexadecimal with the `std::hex` manipulator. The input radix is stored with the formatting flags. The operator also turns off the `skipws` flag. By default, this flag is enabled, which instructs the standard input operators to skip initial white space. By turning this flag off, the input operator does not permit any white space between the pound sign (`#`) and the color value.

When the input function returns, the `ioflags` object is destroyed, and its destructor restores the original formatting flags. Without the magic of RAII, the `operator>>` function would have to restore the flags manually at all four return points, which is burdensome and prone to error.

It makes no sense to copy an `ioflags` object. If you copy it, which object would be responsible for restoring the flags? Thus, the class deletes the copy constructor. If you accidentally write code that would copy the `ioflags` object, the compiler will complain.

RAII is a common programming idiom in C++. The more you learn about C++, the more you will come to appreciate its beauty and simplicity.

As you can see, our examples are becoming more complicated, and it's becoming harder and harder for me to fit entire examples in a single code listing. Your next task is to understand how to separate your code into multiple files, which makes my job and yours much easier. The first step for this new task is to take a closer look at declarations, definitions, and the distinctions between them.

EXPLORATION 40

Declarations and Definitions

Exploration 20 introduced the distinction between declarations and definitions. This is a good time to remind you of the difference and to explore declarations and definitions of classes and their members.

Declaration vs. Definition

Recall that a *declaration* furnishes the compiler with the basic information it needs, so that you can use a name in a program. In particular, a function declaration tells the compiler about the function's name, return type, parameter types, and modifiers, such as `const` and `override`.

A *definition* is a particular kind of declaration that also provides the full implementation details for an entity. For example, a function definition includes all the information of a function declaration, plus the function body. Classes, however, add another layer of complexity, because you can declare or define the class's members independently of the class definition itself. A class definition must declare all of its members. Sometimes, you can also define a member function as part of a class definition (which is the style I've been using so far), but most programmers prefer to declare member functions inside the class and to define the member functions separately, outside of the class definition.

As with any function declaration, a member function declaration includes the return type (possibly with a `virtual` specifier), the function name, the function parameters, and an optional `const` or `override` modifier. If the function is a pure virtual function, you must include the `= 0` token marks as part of the function declaration, and you don't define the function.

The function definition is like any other function definition, with a few exceptions. The definition must follow the declaration—that is, the member function definition must come later in the source file than the class definition that declares the member function. In the definition, omit the `virtual` and `override` specifiers. The function name must start with the class name, followed by the scope operator (`::`) and the function name, so that the compiler knows which member function you are defining. Write the function body the same way you would write it if you provided the function definition inside the class definition. Listing 40-1 shows some examples.

Listing 40-1. Declarations and Definitions of Member Functions

```
class rational
{
public:
  rational();
  rational(int num);
  rational(int num, int den);
  void assign(int num, int den);
  int numerator() const;
  int denominator() const;
  rational& operator=(int num);
private:
  void reduce();
  int numerator_;
  int denominator_;
};

rational::rational()
: rational{0}
{}

rational::rational(int num)
: numerator_{num}, denominator_{1}
{}

rational::rational(int num, int den)
: numerator_{num}, denominator_{den}
{
  reduce();
}

void rational::assign(int num, int den)
{
  numerator_ = num;
  denominator_ = den;
  reduce();
}

void rational::reduce()
{
  assert(denominator_ != 0);
  if (denominator_ < 0)
  {
    denominator_ = -denominator_;
    numerator_ = -numerator_;
  }
  int div{gcd(numerator_, denominator_)};
  numerator_ = numerator_ / div;
  denominator_ = denominator_ / div;
}

int rational::numerator()
const
{
  return numerator_;
}

int rational::denominator()
const
{
  return denominator_;
}
```

```
rational& rational::operator=(int num)
{
  numerator_ = num;
  denominator_ = 1;
  return *this;
}
```

Because each function name begins with the class name, the full constructor name is `rational :: rational`, and member function names have the form `rational :: numerator`, `rational :: operator=`, etc. The C++ term for the complete name is *qualified name*.

Programmers have many reasons to define member functions outside the class. The next section presents one way that functions differ depending on where they are defined, and the next Exploration will focus on this thread in detail.

inline Functions

In Exploration 30, I introduced the `inline` keyword, which is a hint to the compiler that it should optimize speed over size by trying to expand a function at its point of call. You can use `inline` with member functions too. Indeed, for trivial functions, such as those that return a data member and do nothing else, making the function `inline` can improve speed and program size.

When you define a function inside the class definition, the compiler automatically adds the `inline` keyword. If you separate the definition from the declaration, you can still make the function `inline` by adding the `inline` keyword to the function declaration or definition. Common practice is to place the `inline` keyword only on the definition, but I recommend putting the keyword in both places, to help the human reader.

Remember that `inline` is just a hint. The compiler does not have to heed the hint. Modern compilers are becoming better and better at making these decisions for themselves.

My personal guideline is to define one-line functions in the class definition. Longer functions or functions that are complicated to read belong outside the class definition. Some functions are too long to fit in the class definition but are short and simple enough that they should be `inline`. Organizational coding styles usually include guidelines for `inline` functions. For example, directives for large projects may eschew `inline` functions because they increase coupling between software components. Thus, `inline` may be allowed only on a function-by-function basis, when performance measurements demonstrate their need.

Rewrite the rational class from Listing 40-1 to use inline functions judiciously. Compare your solution with that of mine, shown in Listing 40-2.

Listing 40-2. The `rational` Class with `inline` Member Functions

```
class rational
{
public:
  rational(int num) : numerator_{num}, denominator_{1} {}
  rational(rational const&) = default;
  inline rational(int num, int den);
  void assign(int num, int den);
  int numerator() const                 { return numerator_; }
  int denominator() const               { return denominator_; }
  rational& operator=(int num);
private:
  void reduce();
  int numerator_;
  int denominator_;
};
```

```
inline rational::rational(int num, int den)
: numerator_{num}, denominator_{den}
{
  reduce();
}

void rational::assign(int num, int den)
{
  numerator_ = num;
  denominator_ = den;
  reduce();
}

void rational::reduce()
{
  assert(denominator_ != 0);
  if (denominator_ < 0)
  {
    denominator_ = -denominator_;
    numerator_ = -numerator_;
  }
  int div{gcd(numerator_, denominator_)};
  numerator_ = numerator_ / div;
  denominator_ = denominator_ / div;
}

rational& rational::operator=(int num)
{
  numerator_ = num;
  denominator_ = 1;
  return *this;
}
```

Don't agonize over deciding which functions should be inline. When in doubt, don't bother. Make functions inline only if performance measures show that the function is called often and the function call overhead is significant. In all other aspects, I regard the matter as one of aesthetics and clarity: I find one-line functions are easier to read when they are inside the class definition.

Variable Declarations and Definitions

Ordinary data members have declarations, not definitions. Local variables in functions and blocks have definitions, but not separate declarations. This can be a little confusing, but don't be concerned, I'll unravel it and make it clear.

A definition of a named object instructs the compiler to set aside memory for storing the object's value and to generate the necessary code to initialize the object. Some objects are actually sub-objects—not entire objects on their own (entire objects are called *complete* objects in C++ parlance). A sub-object doesn't get its own definition; instead, its memory and lifetime are dictated by the complete object that contains it. That's why a data member or base class doesn't get a definition of its own. Instead, the definition of an object with class type causes memory to be set aside for all of the object's data members. Thus, a class definition contains declarations of data members but not definitions.

You define a variable that is local to a block. The definition specifies the object's type, name, whether it is const, and the initial value (if any). You can't declare a local variable without defining it, but there are other kinds of declarations.

You can declare a local reference as a synonym for a local variable. Declare the new name as a reference in the same manner as a reference parameter, but initialize it with an existing object. If the reference is `const`, you can use any expression (of a suitable type) as the initializer. For a non-`const` reference, you must use an lvalue (remember those from Exploration 21?), such as another variable. Listing 40-3 illustrates these principles.

Listing 40-3. Declaring and Using References

```
#include <iostream>

int main()
{
  int answer{42};    // definition of a named object, also an lvalue
  int& ref{answer};  // declaration of a reference named ref
  ref = 10;          // changes the value of answer
  std::cout << answer << '\n';
  int const& cent{ref * 10}; // declaration; must be const to initialize with expr
  std::cout << cent << '\n';
}
```

A local reference is not a definition, because no memory is allocated, and no initializers are run. Instead, the reference declaration creates a new name for an old object. One common use for a local reference is to create a short name for an object that is obtained from a longer expression, and another is to save a `const` reference to an expression, so that you can use the result multiple times. Listing 40-4 shows a silly program that reads a series of integers into a vector, sorts the data, and searches for all the elements that equal a magic value. It does this by calling the `equal_range` algorithm, which returns a `pair` (first described in Exploration 15) of iterators that delimit a range of equal values.

Listing 40-4. Finding 42 in a Data Set

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <utility>
#include <vector>

int main()
{
  std::vector<int> data{std::istream_iterator<int>(std::cin), std::istream_iterator<int>()};
  std::sort(data.begin(), data.end());
  // Find all values equal to 42
  auto const& range( std::equal_range(data.begin(), data.end(), 42) );
  if (range.first != range.second)
  {
    // Print the range indices only if at least one value is found.
    std::cout << "index of start of range: " << range.first  - data.begin() << '\n';
    std::cout << "index of end of range:   " << range.second - data.begin() << '\n';
  }
  std::cout << "size of range:           " << range.second - range.first << '\n';
}
```

If you define `range` as a local variable instead of declaring it as a reference, the program would work just fine, but it would also make an unneeded copy of the result that `equal_range` returns. In this program, the extra copy is irrelevant and unnoticeable, but in other programs, the cost savings can add up.

What happens if you delete the `const` from the declaration of `range`?

____

____

The result that `equal_range` returns is an rvalue, not an lvalue, so you must use `const` when initializing a reference to that result. Because you are free to modify an object via a non-`const` reference, only lvalue objects are allowed. Usually, the values returned from functions are rvalues, not lvalues, so references must be `const`.

Static Variables

Local variables are *automatic*. This means that when the function begins or a local block (compound statement) is entered, memory is allocated, and the object is constructed. When the function returns or when control exits the block, the object is destroyed, and memory is reclaimed. All automatic variables are allocated on the program stack, so memory allocation and release is trivial and typically handled by the host platform's normal function-call instructions.

Remember that `main()` is like a function and follows many of the same rules as other functions. Thus, variables that you define in `main()` seem to last for the entire lifetime of the program, but they are automatic variables, allocated on the stack, and the compiler treats them the same as it treats any other automatic variables.

The behavior of automatic variables permits idioms such as RAII (see Exploration 39) and greatly simplifies typical programming tasks. Nonetheless, it is not suited for every programming task. Sometimes you need a variable's lifetime to persist across function calls. For example, suppose you need a function that generates unique identification numbers for a variety of objects. It starts a serial counter at 1 and increments the counter each time it issues an ID. Somehow, the function must keep track of the counter value, even after it returns. Listing 40-5 demonstrates one way to do it.

Listing 40-5. Generating Unique Identification Numbers

```
int generate_id()
{
  static int counter{0};
  ++counter;
  return counter;
}
```

The `static` keyword informs the compiler that the variable is not automatic but *static*. The first time the program calls `generate_id()`, the variable `counter` is initialized. The memory is not automatic and is not allocated on the program stack. Instead, all static variables are kept off to the side somewhere, so they don't go away until the program shuts down. When `generate_id()` returns, `counter` is not destroyed and, therefore, retains its value.

Write a program to call `generate_id()` multiple times, to see that it works and generates new values each time you call it. Compare your program with mine, which is shown in Listing 40-6.

Listing 40-6. Calling `generate_id` to Demonstrate Static Variables

```
#include <iostream>

int generate_id()
{
  static int counter{0};
  ++counter;
  return counter;
}
```

```
int main()
{
  for (int i{0}; i != 10; ++i)
    std::cout << generate_id() << '\n';
}
```

You can also declare a variable outside of any function. Because it is outside of all functions, it is not inside any block, thus it cannot be automatic, and so its memory must be static. You don't have to use the `static` keyword for such a variable. **Rewrite Listing 40-6 to declare `counter` outside of the `generate_id` function.** Do not use the `static` keyword. Assure yourself that the program still works correctly. Listing 40-7 shows my solution.

Listing 40-7. Declaring `counter` Outside of the `generate_id` Function

```
#include <iostream>

int counter;

int generate_id()
{
  ++counter;
  return counter;
}

int main()
{
  for (int i{0}; i != 10; ++i)
    std::cout << generate_id() << '\n';
}
```

Unlike automatic variables, all static variables without initializers start out filled with zero, even if the variable has a built-in type. If the class has a custom constructor, the default constructor is then called to initialize static variables of class type. Thus, you don't have to specify an initializer for `counter`, but you can if you want to.

One of the difficulties in working with static variables in C++ is that you have little control over *when* static variables are initialized. The standard offers two basic guarantees:

- Static objects are initialized in the same order as their order of appearance in the file.
- Static objects are initialized before their first use in `main()`, or any function called from `main()`.

Prior to the start of `main()`, however, you have no guarantee that a static object will be initialized when you expect it to be. In practical terms, this means a constructor for a static object should not refer to other static objects, because those other objects may not be initialized yet. All names in C++ are lexically scoped; a name is visible only within its scope. The scope for a name declared within a function is the block that contains the declaration (including the statement header of `for`, `if`, and `while` statements). The scope for a name declared outside of any function is a little trickier. The name of a variable or function is global and can be used only for that single entity throughout the program. On the other hand, you can use it only in the source file where it is declared, from the point of declaration to the end of the file. (The next Exploration will go into more detail about working with multiple source files.)

The common term for variables that you declare outside of all functions is *global variables*. That's not the standard C++ terminology, but it will do for now.

If you declare `counter` globally, you can refer to it and modify it anywhere else in the program, which may not be what you want. It's always best to limit the scope of every name as narrowly as possible. By declaring `counter` inside `generate_id`, you guarantee that no other part of the program can accidentally change its value. In other words, if only one function has to access a static variable, keep the variable's definition local to the function. If multiple functions must share the variable, define the variable globally.

Static Data Members

The `static` keyword has many uses. You can use it before a member declaration in a class to declare a *static data member*. A static data member is one that is not part of any objects of the class but, instead, is separate from all objects. All objects of that class type (and derived types) share a sole instance of the data member. A common use for static data members is to define useful constants. For example: The `std :: string` class has a static data member, `npos`, which roughly means "no position." Member functions return `npos` when they cannot return a meaningful position, such as `find` when it cannot find the string for which it was looking. You can also use static data members to store shared data the same way a globally static variable can be shared. By making the shared variable a data member, however, you can restrict access to the data member using the normal class access levels.

Define a static data member the way you would any other global variable but qualify the member name with the class name. Use the `static` keyword only in the data member's declaration, not in its definition. Because static data members are not part of objects, do not list them in a constructor's initializer list. Instead, initialize static data members the way you would an ordinary global variable, but remember to qualify the member name with the class name. Qualify the name when you use a static data member too. Listing 40-8 shows some simple uses of static data members.

Listing 40-8. Declaring and Defining Static Data Members

```
#include <iostream>

class rational {
public:
  rational();
  rational(int num);
  rational(int num, int den);
  int numerator() const { return numerator_; }
  int denominator() const { return denominator_; }
  // Some useful constants
  static const rational zero;
  static const rational one;
  static const rational pi;
private:
  void reduce(); //get definition from Listing 35-4

  int numerator_;
  int denominator_;
};

rational::rational() : rational{0, 1} {}
rational::rational(int num) : numerator_{num}, denominator_{1} {}
rational::rational(int num, int den)
: numerator_{num}, denominator_{den}
{
  reduce();
}

std::ostream& operator<<(std::ostream& out, rational const& r);

const rational rational::zero{};
const rational rational::one{1};
const rational rational::pi{355, 113};
```

```
int main()
{
  std::cout << "pi = " << rational::pi << '\n';
}
```

A static const data member with an integral type is a little odd, however. Only these data members can have an initializer inside the class definition, as part of the data member's declaration. The value does not change the declaration into a definition, and you still must define the data member outside the class definition. However, by providing a value in the declaration, you can use the static const data member as a constant value elsewhere in the program, anywhere a constant integer is needed.

```
class string {
…
   static size_type const npos{-1};
…
};
```

Like other collection types, string declares size_type as a suitable integer type for representing sizes and indices. The implementation of the string class has to define this data member, but without an initial value.

```
string::size_type const string::npos;
```

Listing 40-9 shows some examples of static data members in a more sophisticated ID generator. This one uses a prefix as part of the IDs it produces and then uses a serial counter for the remaining portion of each ID. You can initialize the prefix to a random number to generate IDs that are unique even across multiple runs of the same program. (The code is not meant to show off a high-quality ID generator, only static data members.) Using a different prefix for every run is fine for production software but greatly complicates testing. Therefore, this version of the program uses the fixed quantity 1. A comment shows the intended code.

Listing 40-9. Using Static Data Members for an ID Generator

```
#include <iostream>

class generate_id
{
public:
  generate_id() : counter_{0} {}
  long next();
private:
  short counter_;
  static short prefix_;
  static short const max_counter_ = 32767;
};

// Switch to random-number as the initial prefix for production code.
// short generate_id::prefix_(static_cast<short>(std::rand()));
short generate_id::prefix_{1};
short const generate_id::max_counter_;

long generate_id::next()
{
  if (counter_ == max_counter_)
```

```
    counter_ = 0;
  else
    ++counter_;
  return static_cast<long>(prefix_) * (max_counter_ + 1) + counter_;
}

int main()
{
  generate_id gen; // Create an ID generator
  for (int i{0}; i != 10; ++i)
    std::cout << gen.next() << '\n';
}
```

Declarators

As you've already seen, you can define multiple variables in a single declaration, as demonstrated in the following:

```
int x{42}, y{}, z{x+y};
```

The entire declaration contains three *declarators*. Each declarator declares a single name, whether that name is for a variable, function, or type. Most C++ programmers don't use this term in everyday conversation, but C++ experts often do. You have to know official C++ terminology, so that if you have to ask for help from the experts, you can understand them.

The most important reason to know about separating declarations from definitions is so you can put a definition in one source file and a declaration in another. The next Exploration shows how to work with multiple source files.

EXPLORATION 41

Using Multiple Source Files

Real programs rarely fit into a single source file, and I know you've been champing at the bit, eager to explore how C++ works with multiple source files that make up a single program. This Exploration shows you the basics. Advanced techniques, such as shared libraries (DLLs, shared objects, etc.), are beyond the scope of this book and sometimes involve compiler-specific features that extend the language beyond the standard. You have to consult your compiler documentation for details.

Multiple Source Files

The basic principle is that you can define any function or global object in any source file. The compiler does not care which file contains what. As long as it has a declaration for every name it needs, it can compile a source file to an object file. (In this unfortunate case of convergent terminology, *object* files are unrelated to *objects* in a C++ program.) To create the final program, you have to link all the object files together. The linker doesn't care which file contains which definition; it simply has to find a definition for every name reference that the compiler generates.

The previous Exploration presented a simple program in Listing 40-9 that generated unique ID numbers. Let's rewrite the program to put the `generate_id` class in one file called `generate_id.cpp` and `main` in another file called `main.cpp`. Listing 41-1 shows the `generate_id.cpp` file.

Listing 41-1. The Definition of `generate_id`'s Members in `generate_id.cpp`

```
class generate_id
{
public:
  generate_id() : counter_{0} {}
  long next();
private:
  short counter_;
  static short prefix_;
  static short const max_counter_{32767};
};

// Switch to random-number as the initial prefix for production code.
// short generate_id::prefix_{static_cast<short>(std::rand())};
short generate_id::prefix_{1};
short const generate_id::max_counter_;
```

```
long generate_id::next()
{
  if (counter_ == max_counter_)
    counter_ = 0;
  else
    ++counter_;
  return static_cast<long>(prefix_) * (max_counter_ + 1) + counter_;
}
```

You can compile the `generate_id.cpp` file but not link it. To create a valid C++ program, you must have a `main` function, which is presented in Listing 41-2. Because `main` makes use of the `generate_id` class, it requires the class definition but not the definitions of the class's members.

Listing 41-2. The `main` Function for the ID-generating Program in the `main.cpp` File

```
#include <iostream>

class generate_id
{
public:
  generate_id() : counter_{0} {}
  long next();
private:
  short counter_;
  static short prefix_;
  static short const max_counter_{32767};
};

int main()
{
  generate_id gen{};           // Create an ID generator
  for (int i{0}; i != 10; ++i)
    std::cout << gen.next() << '\n';
}
```

Now compile the two source files and link them together to produce a working C++ program. An IDE takes care of the details for you, provided both source files are part of the same project. If you are using command-line tools, you can invoke the same compiler, but instead of listing source file names on the command line, list only the object file names. Alternatively, you can compile and link at the same time, by listing all the source file names in one compilation. Verify that the program's behavior is identical to that from Listing 40-9.

That's the basic idea, but the details, of course, are a little trickier. For the remainder of this Exploration, we'll take a closer look at those details.

Declarations and Definitions

At first glance, you will notice that both source files contain an identical definition of the `generate_id` class. That immediately raises the question, what happens if the definition changes in one file but not in the other? Let's find out. Rename the `next()` function to `next_id()`, as shown in Listing 41-3.

Listing 41-3. Renaming a Member Function Only in `main.cpp`

```
#include <iostream>

class generate_id
{
public:
  generate_id() : counter_{0} {}
  long next_id();
private:
  short counter_;
  static short prefix_;
  static short const max_counter_{32767};
};

int main()
{
  generate_id gen{};           // Create an ID generator
  for (int i{0}; i != 10; ++i)
    std::cout << gen.next_id() << '\n';
}
```

Leave the other source file alone. **What do you expect to happen?**

____

____

Compile and link the program. **What actually happens?**

____

____

When compiling `main.cpp`, the compiler sees the declaration of `next_id` and the call to `next_id`. As far as it can tell, everything is just fine. The linker, however, sees a call to `next_id` in the `main` object file but no definition of `next_id` in the `generate_id` object file. Thus, the linker issues an error message and refuses to create the executable program file:

```
list4103.o: In function 'main':
list4103.cpp:18: undefined reference to 'generate_id::next_id()'
collect2: error: ld returned 1 exit status
```

A more subtle change is to add a new data member before `counter`, as shown in Listing 41-4.

Listing 41-4. Adding a New Data Member Only in `main.cpp`

```
#include <iostream>
#include <ostream>

class generate_id
{
public:
  generate_id() : counter_{0} {}
  long next();
private:
  int unused_;
  short counter_;
  static short prefix_;
  static short const max_counter_{32767};
};

int main()
{
  generate_id gen{};           // Create an ID generator
  for (int i{0}; i != 10; ++i)
    std::cout << gen.next() << '\n';
}
```

Leaving the other source file untouched, **what do you expect to happen?**

____

____

Compile, link, and run the program. **What actually happens?**

____

____

Remember that the program should produce identical results. The exact numeric sequence should be the same between this program and earlier incarnations. The problem is that you have stumbled once again into undefined behavior.

Even though the main function thinks the `unused_` data member is indeed unused, the `next()` function's idea of the `counter_` data member happens to be at the same address as the `unused_` data member. This is bad—really bad. But if you weren't watching closely, you might have missed it. Although anything is possible, most likely the program generated a sequence of unique IDs, and if you hadn't looked at the actual values but simply used the `generate_id` class in a program, you might never have noticed. At least, not until you ported the program to a different environment. A different compiler or linker may be able to detect this error. A different operating system may cause the program to crash. Once you enter the realm of undefined behavior, anything goes.

A class or function definition must be identical in all files, or else the results are undefined. The compiler and linker are not required to detect this kind of error.

You have been warned.

To avoid these kinds of problems, you have to host the definition of the `generate_id` class in a single file and somehow use that file in every source file that makes use of the class. This way, you can assure yourself that every source file is using the same definition of `generate_id`.

You must write your own `#include` file.

#include Files

Remember that the compiler needs only a function's declaration, not its definition, in order to call the function. If you put the declaration in its own file, then #include that file in every source file that calls the function, you ensure that the compiler sees the same declaration every time, allowing it to generate the correct code for calling the function. The same is true for a class. Put only the class definition in its own file, and then #include the file in every source file that needs it. Listing 41-5 shows you the #include file that contains the generate_id class. Common conventions for #include files are to use extensions such as .h or .hpp (for header or C++ header) or, sometimes, .hh or .hxx. Some files have to preserve compatibility with C, and these typically use .h as the extension. For all other files, I prefer .hpp, because it is a nice parallel to the extension .cpp that I use for C++ source files. Thus, name this file generate_id.hpp. If you use .cc for source files, you may like .hh for header files, ditto for .cxx and .hxx.

Listing 41-5. The #include File for the generate_id Class

```
/// Class for generating a unique ID number.
class generate_id
{
public:
  generate_id() : counter_{0} {}
  long next();
private:
  short counter_;
  static short prefix_;
  static short const max_counter_{32767};
};
```

To use the generate_id.hpp file, you have to #include it in the source files, but use double quotes instead of angle brackets. Listing 41-6 displays the new version of generate_id.cpp.

Listing 41-6. Rewrite of generate_id.cpp to #include the generate_id.hpp File

```
#include "generate_id.hpp"

// Switch to random-number as the initial prefix for production code.
// short generate_id::prefix_{static_cast<short>(std::rand())};
short generate_id::prefix_{1};
short const generate_id::max_counter_;

long generate_id::next()
{
  if (counter_ == max_counter_)
    counter_ = 0;
  else
    ++counter_;
  return prefix_ * (max_counter_ + 1) + counter_;
}
```

Rewrite main.cpp similarly. Compile both source files and link the resulting object files to create your program (or let the IDE do it for you). Make sure the program behaves the same as it did originally. Compare your rewrite of main.cpp with mine, which is presented in Listing 41-7.

Listing 41-7. Rewriting `main.cpp` to `#include` the `generate_id.hpp` File

```
#include <iostream>

#include "generate_id.hpp"

int main()
{
  generate_id gen{};           // Create an ID generator
  for (int i{0}; i != 10; ++i)
    std::cout << gen.next() << '\n';
}
```

Quotes and Brackets

Now you're wondering why I told you to use quotes instead of angle brackets for the `#include` directives. The difference is that you should use angle brackets only for the standard library and system headers, although some third-party libraries recommend the use of angle brackets too. Use double quotes for everything else. The C++ standard is deliberately vague and recommends that angle brackets be used for headers that are provided with the system and quotes be used for other headers. Vendors of add-on libraries have all taken different approaches concerning naming their library files and whether they require angle brackets or double quotes.

For your own files, the important aspect is that the compiler must be able to find all your `#include` files. The easiest way to do that is to keep them in the same directory or folder as your source files. As your projects become larger and more complex, you probably will want to move all the `#include` files to a separate area. In this case, you have to consult your compiler documentation, to learn how to inform the compiler about that separate area. Users of g++ and other UNIX and UNIX-like command-line tools typically use the `-I` (capital letter *I*) option. Microsoft's command-line compiler uses `/I`. IDEs have a project option with which you can add a directory or folder to the list of places to search for `#include` files.

For many compilers, the only difference between angle brackets and quotes is where it looks for the file. A few compilers have additional differences that are specific to that compiler.

In a source file, I like to list all the standard headers together, in alphabetical order, and list them first, followed by the `#include` files that are specific to the program (also in alphabetical order). This organization makes it easy for me to determine whether a source file `#includes` a particular header and helps me add or remove `#include` directives as needed.

Nested `#include` Directives

One `#include` file can `#include` another. For example, consider the `vital_stats` class (similar to the `record` class in Listing 34-1, for recording a person's vital statistics, including body-mass index) in Listing 41-8.

Listing 41-8. The `vital_stats` Class to Record a Person's Vital Statistics

```
#include <istream>
#include <ostream>
#include <string>

class vital_stats
{
public:
  vital_stats() : height_{0}, weight_{0}, bmi_{0}, sex_{'?'}, name_{}
  {}
  bool read(std::istream& in, int num);
  void print(std::ostream& out, int threshold) const;

private:
  int compute_bmi() const; ///< Return BMI, based on height_ and weight_
  int height_;             ///< height in centimeters
  int weight_;             ///< weight in kilograms
  int bmi_;                ///< Body-mass index
  char sex_;               ///< 'M' for male or 'F' for female
  std::string name_;       ///< Person's name
};
```

Because the `vital_stats` class uses `std::string`, the `vital_stats.hpp` file should `#include <string>`. Similarly, `std::istream` is defined in `<istream>` and `std::ostream` in `<ostream>`. By adding all the necessary `#include` directives to the `vital_stats.hpp` file, you remove one burden from the programmer who makes use of `vital_stats.hpp`.

The standard library headers work the same way. Any particular header may `#include` other headers. An implementation of the standard library is free to `#include` any, all, or none of the other standard headers. Thus, suppose you `#include <string>` and forget the I/O stream headers. With one compiler and library, your project may compile and run successfully, because that particular implementation of `<string>` happens to `#include <istream>` and `<ostream>`. You never notice your mistake until you move the source code to a different platform. This other implementation may work differently. Suddenly and unexpectedly, your compiler issues a slew of error messages for source code that worked perfectly on the first platform. The mistake is easy to fix and also easy to avoid, by checking all your files to be sure each file `#include`s all the headers it needs.

One consequence of headers including other headers is that any single header can be included many times in a single source file. For example, suppose your source file includes `<string>`, but so does `generate_id.hpp`. This means your source file includes the same header, `<string>`, more than once. In the case of the standard library headers, including the same header more than once is harmless. What about your files? **What would happen if *generate_id.cpp* were to #include `"generate_id.hpp"` more than once?**

____

____

Try it. **What happens?**

____

____

C++ does not allow you to define a class, function, or object more than once in the same source file, and `generate_id.hpp` contains a class definition. Thus, if you `#include` that file more than once, the compiler issues an error message. The next section explains how to prevent this problem.

Include Guards

You cannot define the same class, function, or object more than once in a source file. On the other hand, you cannot prevent anyone from including your header more than once. Somehow, you must ensure that the compiler sees the definitions only once, even if a source file includes your header more than once. Fortunately, C++ offers additional # directives to help you.

The most widely used idiom is to pick a name to uniquely identify your header, such as `VITAL_STATS_HPP_` or `GENERATE_ID_HPP_`, and then use that name to control the compilation of your header. Use the following two lines as the first two lines of the header file:

```
#ifndef GENERATE_ID_HPP_
#define GENERATE_ID_HPP_
```

Next, use the following as the last line of the header file:

```
#endif
```

Convention is to use all capital letters, because no other names in your program should be made up of all capital letters. I use the file name but change dot (`.`) and other characters that aren't allowed in C++ identifiers into underscores (`_`). I use a trailing underscore to further ensure that I avoid collisions with other names. (Remember that names beginning with an underscore are not allowed, nor are names with two adjacent underscores.)

The `#ifndef` directive means "if not defined." The directive controls whether the compiler compiles the source code normally or skips the entire section of the file. If the name is not defined, compilation proceeds normally. If it is defined, the compiler skips rapidly over the file until it finds `#endif`, thereby ignoring the definitions that you want to protect.

The `#define` directive defines a name. Thus, the first time the compiler processes this header, the name is not defined, the `#ifndef` condition is true, and the compiler processes the definitions normally. Among the code it processes normally is the `#define`. If the file is included more than once, the second and subsequent times the name is defined, the `#ifndef` directive is false, so the compiler skips the bulk of the file and does not try to compile the definitions more than once.

BAD ADVICE

A few books and programmers recommend putting the conditional directives in the file that does the `#include`-ing. For example, they maintain that `main.cpp` should begin like this:

```
#ifndef GENERATE_ID_HPP_
#include "generate_id.hpp"
#endif
```

They are wrong. The name `GENERATE_ID_HPP_` is an internal, private detail of the included file. You should never use it outside of that file, and no one else should ever use that name. The purported reason to export the guard name and use it outside of the file is to improve compilation speed by eliminating the compiler's need to open and process the file that would otherwise be included. With modern compilers, the improvement is negligible. A better solution is for you to use pre-compiled headers.

With large, complicated programs, much of the compiler's time and effort is spent compiling included files. In other words, the compiler spends a lot of its time recompiling the same definitions over and over again. The idea of using pre-compiled headers is that the compiler saves some information about the definitions it sees in an included file. When you include the same file in other source files, the compiler fetches its pre-compiled data and saves time by avoiding a recompilation of the same definitions. This compiler hack is completely unrelated to the C++ language and is purely an artifact of the compiler program. Check your compiler's documentation, to learn how to set up and use pre-compiled headers.

Forward Declarations

The `<istream>` header contains the full declaration of `std::istream` and other, related declarations, and `<ostream>` declares `std::ostream`. These are large classes in large headers. Sometimes, you don't need the full class declarations. For example, in `vital_stats.hpp`, the only use of `std::ostream` is to declare a reference parameter to the `print()` member function. The compiler doesn't have to know all the details of `std::ostream`; it merely needs to know that `std::ostream` is a class.

The header `<iosfwd>` is a small header that declares the names `std::istream`, `std::ostream`, etc., without providing the complete class declarations. Thus, you can reduce compilation time for any file that includes `vital_stats.hpp` by changing `<istream>` and `<ostream>` to `<iosfwd>`.

You can do the same for your own classes by declaring the class name after the `class` keyword, with nothing else describing the class:

```
class ostream;
```

This is known as a *forward* declaration. You can use a forward declaration when the compiler has to know a name is a class but doesn't have to know the size of the class or any of the class's members. A common case is using a class solely as a reference function parameter.

If your header uses `<iosfwd>` or other forward declarations, be sure to include the full class declarations (e.g., `<iostream>`) in the .cpp source file.

Documentation

Recall the Doxygen tool from Exploration 27. With headers, you now face the problem that you have two places to document certain entities: the declaration in the header file, and the definition in the source file. One option is to put the documentation in the header file. The documentation is usually aimed at the user of the entity, and the header is the proper place to document the interface.

Another option is to document the public interface in the header file and the private implementation details in the source file. The Doxygen tool cannot merge documentation from two sources for a single entity. Instead, you should take advantage of its conditional processing features to compile a set of interface documentation and a separate set of implementation documentation. Listing 41-9 shows the final version of the `vital_stats.hpp` header file.

Listing 41-9. The `vital_stats.hpp` Header File

```
#ifndef VITAL_STATS_HPP_
#define VITAL_STATS_HPP_

#include <iosfwd>
#include <string>

class vital_stats
{
public:
  /// Constructor. Initialize everything to zero or other "empty" value.
  vital_stats() : height_{0}, weight_{0}, bmi_{0}, sex_{'?'}, name_{}
  {}

  /// Get this record, overwriting the data members.
  /// Error-checking omitted for brevity.
  /// @param in the input stream
  /// @param num a serial number, for use when prompting the user
  /// @return true for success or false for eof or input failure
  bool read(std::istream& in, int num);

  /// Print this record to @p out.
  /// @param out the output stream
  /// @param threshold mark records that have a BMI >= this threshold
  void print(std::ostream& out, int threshold) const;

  /// Return the BMI.
  int bmi() const { return bmi_; }
  /// Return the height in centimeters.
  int height() const { return height_; }
  /// Return the weight in kilograms.
  int weight() const { return weight_; }
  /// Return the sex: 'M' for male or 'F' for female
  char sex() const { return sex_; }
  /// Return the person's name.
  std::string const& name() const { return name_; }

private:
  /// Return BMI, based on height_ and weight_
  /// This is called only from read().
  int compute_bmi() const;
  int height_;             ///< height in centimeters
  int weight_;             ///< weight in kilograms
  int bmi_;                ///< Body-mass index
  char sex_;               ///< 'M' for male or 'F' for female
  std::string name_;       ///< Person's name
};

#endif
```

Listing 41-10 presents the implementation in `vital_stats.cpp`.

Listing 41-10. The `vital_stats.cpp` Source File

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <locale>
#include <string>

#include "vital_stats.hpp"

/// Skip the rest of the input line.
/// @param in the input stream
void skip_line(std::istream& in)
{
  in.ignore(std::numeric_limits<int>::max(), '\n');
}
```

```
int vital_stats::compute_bmi()
const
{
   return static_cast<int>(weight_ * 10000 / (height_ * height_) + 0.5);
}

bool vital_stats::read(std::istream& in, int num)
{
  std::cout << "Name " << num << ": ";
  if (not std::getline(in, name_))
    return false;

  std::cout << "Height (cm): ";
  if (not (in >> height_))
    return false;
  skip_line(in);

  std::cout << "Weight (kg): ";
  if (not (in >> weight_))
    return false;
  skip_line(in);

  std::cout << "Sex (M or F): ";
  if (not (in >> sex_))
    return false;
  skip_line(in);
  sex_ = std::toupper(sex_, std::locale{});

  bmi_ = compute_bmi();
 return true;
}

void vital_stats::print(std::ostream& out, int threshold)
const
{
  out << std::setw(6) << height_
      << std::setw(7) << weight_
      << std::setw(3) << sex_
      << std::setw(6) << bmi_;
  if (bmi_ >= threshold)
    out << '*';
  else
    out << ' ';
  out << ' ' << name_ << '\n';
}
```

extern Variables

If you define a global variable, and you want to use it in multiple files, you need a declaration in every file that uses the variable. The most common use for global objects is for useful constants. For example, one of the omissions many scientific programmers notice when they first begin to use C++ is that the standard math header, `<cmath>`, lacks a definition for π.

Suppose you decide to remedy this oversight by creating your own header file, `math.hpp`. This header file contains the declarations for `pi` and other useful constants. Declare a global variable by using the `extern` keyword, followed by the variable's type and name.

Write `math.hpp`, with proper `#include` guards and a declaration for a double constant named `pi`. Compare your file with Listing 41-11.

Listing 41-11. Simple Header for Math Constants

```
#ifndef MATH_HPP_
#define MATH_HPP_

extern double pi;

#endif
```

One source file in the project must define `pi`. Name the file `math.cpp`. **Write `math.cpp`.** Compare your file with Listing 41-12.

Listing 41-12. Definitions of Math Constants

```
#include "math.hpp"

// More digits that typical implementations of double support.
double pi{3.14159265358979323846264338327};
```

`inline` Functions

Typically, you declare functions in a header file and define them in a separate source file, which you then link with your program. This is true for free functions and for member functions. Thus, most member functions are defined separately from the class.

Inline functions follow different rules than ordinary functions. Any source file that calls an inline function requires the function's definition. Each source file that uses an inline function must have no more than one definition of that inline function, and every definition in the program must be the same. Thus, the rule for functions in header files is slightly more complicated: the header file contains declarations for non-inline functions and definitions for inline functions. The separate source file defines only the non-inline functions.

Inline functions have their uses, but they also have some significant drawbacks.

- *Increased compilation time.* With inline functions, the header file is larger, and the compiler does more work for every compilation in the project. The problem grows as the number of source files grows.
- *More frequent recompilation.* If you modify a function body, only that body really needs to be recompiled. However, if the function body is in a header file, you end up recompiling every source file that includes that header. By putting function bodies in a separate source file, only that file needs to be recompiled, and you can save time. In large projects, the amount of time you save can be significant.

Separating function declarations and definitions makes sense in real programs, but it complicates this book. Instead of a single code listing, sometimes I will need multiple code listings. Nonetheless, I'll do my best to demonstrate good programming practices in the judicious use of inline functions and the separation of declarations and definitions.

One-Definition Rule

The compiler enforces the rule that permits one definition of a class, function, or object per source file. Another rule is that you can have only one definition of a function or global object in the entire program. You can define a class in multiple source files, provided the definition is the same in all source files.

As mentioned in the previous section, inline functions follow different rules than ordinary functions. You can define an inline function in multiple source files. Each source file must have no more than one definition of the inline function, and every definition in the program must be the same.

These rules are collectively known as the One-Definition Rule (ODR).

The compiler enforces the ODR within a single source file. However, the standard does not require a compiler or linker to detect any ODR violations that span multiple source files. If you make such a mistake, the problem is all yours to find and fix.

Imagine that you are maintaining a program, and part of the program is the header file shown in Listing 41-13.

Listing 41-13. The Original `point.hpp` File

```
#ifndef POINT_HPP_
#define POINT_HPP_
class point
{
public:
  point() : point{0, 0} {}
  point(int x, int y) : x_{x}, y_{y} {}
  int x() const { return x_; }
  int y() const { return y_; }
private:
  int y_, x_;
};
#endif // POINT_HPP_
```

The program works just fine. One day, however, you upgrade compiler versions, and when recompiling the program, the new compiler issues a warning, such as the following, that you've never seen before:

```
point.hpp: In constructor 'point::point()':
point.hpp:13: warning: 'point::x_' will be initialized after
point.hpp:13: warning:   'int point::y_'
point.hpp:8: warning:   when initialized here
```

The problem is that the order of the data member declarations is different from the order of the data members in the constructors' initializer lists. It's a minor error, but one that can lead to confusion, or worse, in more complicated classes. It's a good idea to ensure the orders are the same. Suppose, you decide to fix the problem by reordering the data members.

Then you recompile the program, but the program fails in mysterious ways. Some of your regression tests pass and some fail, including trivial tests that have never failed in the past.

What went wrong?

____

____

With such limited information, you can't determine for certain what went wrong, but the most likely scenario is that the recompilation failed to capture all the source files. Some part of the program (not necessarily the part that is failing) is still using the old definition of the `point` class, and other parts of the program use the new definition. The program fails to adhere to the ODR, resulting in undefined behavior. Specifically, when the program passes a `point` object from one part of the program to another, one part of the program stores a value in `x_`, and another part reads the same data member as `y_`.

This is only one small example of how ODR violations can be both subtle and terrible at the same time. By ensuring that all class definitions are in their respective header files, and that any time you modify a header file you recompile all dependent source files, you can avoid most accidental ODR violations.

Now that you have the tools needed to start writing some serious programs, it's time to embark on some more advanced techniques. The next Exploration introduces function objects—a powerful technique for using the standard algorithms.

EXPLORATION 42

Function Objects

Classes have many, many uses in C++ programs. This Exploration introduces one powerful use of classes to replace functions. This style of programming is especially useful with the standard algorithms.

The Function Call Operator

The first step is to take a look at an unusual "operator," the function call operator, which lets an object behave as a function. Overload this operator the same way you would any other. Its name is `operator()`. It takes any number of parameters and can have any return type. Listing 42-1 shows another iteration of the `generate_id` class (last seen in Listing 41-5), this time replacing the `next()` member function with the function call operator. In this case, the function has no parameters, so the first set of empty parentheses is the operator name, and the second set is the empty parameter list.

Listing 42-1. Rewriting `generate_id` to Use the Function Call Operator

```
#ifndef GENERATE_ID_HPP_
#define GENERATE_ID_HPP_

/// Class for generating a unique ID number.
class generate_id
{
public:
  generate_id() : counter_{0} {}
  long operator()();
private:
  short counter_;
  static short prefix_;
  static short const max_counter_{32767};
};

#endif
```

Listing 42-2 displays the implementation of the function call operator (and `prefix_`, which also requires a definition).

Listing 42-2. Implementation of the `generate_id` Function Call Operator

```
#include "generate_id.hpp"

short generate_id::prefix_{1};

long generate_id::operator()()
{
  if (counter_ == max_counter_)
    counter_ = 0;
  else
    ++counter_;
  return static_cast<long>(prefix_) * (max_counter_ + 1) + counter_;
}
```

In order to use the function call operator, you must first declare an object of the class type, then use the object name as though it were a function name. Pass arguments to this object the way you would to an ordinary function. The compiler sees the use of the object name as a function and invokes the function call operator. Listing 42-3 shows a sample program that uses a `generate_id` function call operator to generate ID codes for new library works. (Remember the `work` class from Exploration 38? Collect `work` and its derived classes into a single file, add the necessary `#include` directives, and include guard. Call the resulting file `library.hpp`. Or download a complete `library.hpp` from the book's web site.) Assume that `int_to_id` converts an integer identification into the string format that `work` requires and that `accession` adds a `work`-derived object to the library's database.

Listing 42-3. Using a `generate_id` Object's Function Call Operator

```
#include <iostream>

#include "generate_id.hpp"
#include "library.hpp"

bool get_movie(std::string& title, int& runtime)
{
  std::cout << "Movie title: ";
  if (not std::getline(std::cin, title))
    return false;
  std::cout << "Runtime (minutes): ";
  if (not (std::cin >> runtime))
    return false;
  return true;
}

int main()
{
  generate_id gen{};           // Create an ID generator
  std::string title{};
  int runtime{};
  while (get_movie(title, runtime))
  {
    movie m(int_to_id(gen()), title, runtime);
    accession(m);
  }
}
```

Function Objects

A *function object* or *functor* is an object of class type for a class that overloads the function call operator. Informally, programmers sometimes also speak of the class as a “function object,” with the understanding that the actual function objects are the variables defined with that class type.

C++ 03 programs often used functors, but C++ 11 has lambdas, which are much easier to read and write. (Recall lambdas from Exploration 23?) So what do functors offer that lambdas lack? To answer that question, consider the following problem.

Suppose you need a vector that contains integers of increasing value. For example, a vector of size 10 would contain the values 1, 2, 3, . . . , 8, 9, 10. The `std::generate` algorithm takes an iterator range and calls a function or functor for each element of the range, assigning the result of the functor to successive elements. **Write a lambda to use as the final argument to `std::generate`**. (Construct a vector of a specific size by passing the desired size to a constructor. Remember to use parentheses instead of curly braces to call the right constructor.) Compare your solution with mine in Listing 42-4.

Listing 42-4. The Main Program for Generating Successive Integers

```
#include <algorithm>
#include <vector>

int main()
{
  std::vector<int> vec(10);
  int state;
  std::generate(vec.begin(), vec.end(), [&state]() { return ++state; });
}
```

Okay, that was easy, but the solution is not very general. The lambda cannot be reused anywhere else. It needs the `state` variable, and you can have only one such lambda per state variable. Can you think of a way to write a lambda such that you can have more than one generator, each with its own state? That’s much harder to do with lambdas, but easy with functors. **Write a functor class to generate successive integers, so the functor can be used with the generate algorithm**. Name the class `sequence`. The constructor takes two arguments: the first specifies the initial value of the sequence, and the second is the increment. Each time you call the function call operator, it returns the generator value, then increments that value, which will be the value returned on the next invocation of the function call operator. Listing 42-5 shows the main program. Write your solution in a separate file, `sequence.hpp`, using only inline functions (so you don’t have to compile a separate `sequence.cpp` source file).

Listing 42-5. The Main Program for Generating Successive Integers

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

#include "sequence.hpp"

int main()
{
  int size{};
  std::cout << "How many integers do you want? ";
  std::cin >> size;
  int first{};
  std::cout << "What is the first integer? ";
  std::cin >> first;
  int step{};
  std::cout << "What is the interval between successive integers? ";
  std::cin >> step;

  std::vector<int> data(size);
  // Generate the integers to fill the vector.
  std::generate(data.begin(), data.end(), sequence(first, step));

  // Print the resulting integers, one per line.
  std::copy(data.begin(), data.end(),
            std::ostream_iterator<int>(std::cout, "\n"));
}
```

Compare your solution with mine, shown in Listing 42-6.

Listing 42-6. The `sequence.hpp` File

```
#ifndef SEQUENCE_HPP_
#define SEQUENCE_HPP_

/// Generate a sequence of integers.
class sequence
{
public:
  /// Construct the functor.
  /// @param start the first value the generator returns
  /// @param step increment the value by this much for each call
  sequence(int start, int step ) : value_{start}, step_{step} {}
  sequence(int start) : sequence{start, 1} {}
  sequence() : sequence{0} {}

  /// Return the current value in the sequence, and increment the value.
  int operator()()
  {
    int result(value_);
    value_ = value_ + step_;
    return result;
  }
private:
  int value_;
  int const step_;
};

#endif
```

The `generate` algorithm has a partner, `generate_n`, which specifies an input range with an iterator for the start of the range and an integer for the size of the range. The next Exploration examines this and several other useful algorithms.

EXPLORATION 43

Useful Algorithms

The standard library includes a suite of functions, which the library calls *algorithms*, to simplify many programming tasks that involve repeated application of operations over data ranges. The data can be a container of objects, a portion of a container, values read from an input stream, or any other sequence of objects that you can express with iterators. I've introduced a few algorithms when appropriate. This Exploration takes a closer look at a number of the most useful algorithms.

Searching

The standard algorithms include many flavors of searching, divided into two broad categories: linear and binary. The linear searches examine every element in a range, starting from the first and proceeding to subsequent elements until reaching the end (or the search ends because it is successful). The binary searches require the elements be sorted in ascending order, using the < operator, or according to a custom predicate, that is, a function, functor, or lambda that returns a Boolean result.

Linear Search Algorithms

The most basic linear search is the `find` function. It searches a range of read iterators for a value. It returns an iterator that refers to the first matching element in the range. If `find` cannot find a match, it returns a copy of the end iterator. Listing 43-1 shows an example of its use. The program reads integers into a vector, searches for the value 42, and if found, changes that element to 0.

Listing 43-1. Searching for an Integer

```
#include <algorithm>
#include <iostream>

#include "data.hpp"

int main()
{
  intvector data{};
  read_data(data);
  write_data(data);
  auto iter(std::find(data.begin(), data.end(), 42));
  if (iter == data.end())
    std::cout << "Value 42 not found\n";
  else
  {
    *iter = 0;
    std::cout << "Value 42 changed to 0:\n";
    write_data(data);
  }
}
```

Listing 43-2 shows the *data.hpp* file, which provides a few utilities for working with vectors of integers. Most of the examples in this Exploration will #include this file.

Listing 43-2. The data.hpp File to Support Integer Data

```
#ifndef DATA_HPP_
#define DATA_HPP_

#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

/// Convenient shorthand for a vector of integers.
typedef std::vector<int> intvector;

/// Convenient shorthand for an intvector's iterator.
typedef intvector::iterator intvec_iterator;

/// Read a series of integers from the standard input into @p data,
/// overwriting @p data in the process.
/// @param[in,out] data a vector of integers
inline void read_data(intvector& data)
{
  data.clear();
  data.insert(data.begin(), std::istream_iterator<int>(std::cin),
                            std::istream_iterator<int>());
}

/// Write a vector of integers to the standard output. Write all values on one
/// line, separated by single space characters, and surrounded by curly braces,
/// e.g., { 1 2 3 }.
/// @param data a vector of integers
inline void write_data(intvector const& data)
{
  std::cout << "{ ";
  std::copy(data.begin(), data.end(),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout << "}\n";
}

#endif
```

A companion to the find algorithm is find_if. Instead of searching for a matching value, find_if takes a predicate function or function object (from now on, I will write *functor* to indicate a free function, a function object, or a lambda). It calls the functor for every element in the range, until the functor returns true (or any value that can be converted automatically to true, such as a nonzero numeric value). If the functor never returns true, find_if returns the end iterator.

Every search algorithm comes in two forms. The first compares items using an operator (== for linear searches and < for binary searches). The second form uses a caller-supplied functor instead of the operator. For most algorithms, the functor is an additional argument to the algorithm, so the compiler can distinguish the two forms. In a few cases, both forms take the same number of arguments, and the library uses distinct names, because the compiler could not otherwise distinguish between the two forms. In these cases, the functor form has _if added to the name, such as find and find_if.

Suppose you want to search a vector of integers, not for a single value, but for any value that falls within a certain range. You can write a custom predicate to test a hard-coded range, but a more useful solution is to write a general-purpose functor that compares an integer against any range. Use this functor by supplying the range limits as argument to the constructor. **Is this best implemented as a free function, function object, or lambda?** _________________________ Because it must store state, I recommend writing a functor.

A lambda is good when you need to search for specific values, but a functor is easier if you want to write a generic comparator that can store the limits. **Write the intrange functor**. The constructor takes two int arguments. The function call operator takes a single int argument. It returns true, if the argument falls within the inclusive range specified in the constructor, or false, if the argument lies outside the range.

Listing 43-3 shows my implementation of intrange. As a bonus, I decided to allow the caller to specify the range limits in either order. That way, I neatly avoid the issue of error-checking and error-handling, if the caller tries to use a meaningless range, such as [10, 0].

Listing 43-3. Functor intrange to Generate Integers in a Certain Range

```
#ifndef INTRANGE_HPP_
#define INTRANGE_HPP_

#include <algorithm>

/// Check whether an integer lies within an inclusive range.
class intrange
{
public:
  inline intrange(int low, int high);
  inline bool operator()(int test) const;
private:
  int const low_;
  int const high_;
};

/// Construct an integer range.
/// If the parameters are in the wrong order,
/// swap them to the right order.
/// @param low the lower bound of the inclusive range
/// @param high the upper bound of the inclusive range
inline intrange::intrange(int low, int high)
: low_{std::min(low, high)}, high_{std::max(low, high)}
{}
```

```
/// Check whether a value lies within the inclusive range.
/// @param test the value to test
inline bool intrange::operator()(int test)
const
{
  return test >= low_ and test <= high_;
}

#endif
```

The < operator form of the `std::min` function takes two arguments and returns the smaller. The `std::max` function also takes two arguments and returns the larger. Both functions compare their arguments with the < operator. As with other algorithms, you can call a functor form of both functions, passing a comparison functor to use instead of the < operator. The types of the first two arguments must be the same, and the return type matches that of the arguments. The `mixmax` function returns a `std::pair` (introduced in Exploration 15) of the minimum and maximum.

Write a test program that reads integers from the standard input and then uses `find_if` and `intrange` to find the first value that lies within the range [10, 20]. Compare your solution with mine in Listing 43-4.

Listing 43-4. Using `find_if` and `intrange` to Find an Integer That Lies Within a Range

```
#include <algorithm>
#include <iostream>

#include "data.hpp"
#include "intrange.hpp"

int main()
{
  intvector data{};
  read_data(data);
  write_data(data);
  auto iter(std::find_if(data.begin(), data.end(), intrange{10, 20}));
  if (iter == data.end())
    std::cout << "No values in [10,20] found\n";
  else
    std::cout << "Value " << *iter << " in range [10,20].\n";
}
```

A few of the following examples generate random data and apply algorithms to the data. The standard library has a rich, complicated library for generating pseudo-random numbers. The details of this library are beyond the scope of this book. Only the mathematically adventuresome should crack open the details of the `<random>` header. For your convenience, Listing 43-5 presents the `randomint.hpp` header, which defines the `randomint` class, which generates random integers in a caller-supplied range.

Listing 43-5. Generating Random Integers

```
#ifndef RANDOMINT_HPP_
#define RANDOMINT_HPP_

#include <algorithm>
#include <random>
```

```
/// Generate uniformly distributed random integers in a range.
class randomint
{
public:
  typedef std::default_random_engine::result_type result_type;

  /// Construct a random-number generator to produce numbers in the range [<tt>low</tt>, <tt>high</tt>].
  /// If @p low > @p high the values are reversed.
  randomint(result_type low, result_type high)
     // std::random_device uses a system-dependent generation of randomness
      // to seed the pseudo-random-number generator.
  : prng_{std::random_device{}()},
    distribution_{std::min(low, high), std::max(low, high)}
  {}

  /// Generate the next random number generator.
  result_type operator()()
  {
     return distribution_(prng_);
  }

private:
  // implementation-defined pseudo-random-number generator
  std::default_random_engine prng_;
  // Map random numbers to a uniform distribution.
  std::uniform_int_distribution<result_type> distribution_;
};
#endif
```

The search function is similar to find, except it searches for a matching sub-range. That is, you supply an iterator range to search and an iterator range to match. The search algorithm looks for the first occurrence of a sequence of elements that equals the entire match range. Listing 43-6 shows a silly program that generates a large vector of random integers in the range 0 to 9 and then searches for a sub-range that matches the first four digits of π.

Listing 43-6. Finding a Sub-range That Matches the First Four Digits of π

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

#include "data.hpp"
#include "randomint.hpp"

int main()
{
  intvector pi{ 3, 1, 4, 1 };
  intvector data(10000);
  // The randomint functor generates random numbers in the range [0, 9].
  std::generate(data.begin(), data.end(), randomint{0, 9});

  auto iter(std::search(data.begin(), data.end(), pi.begin(), pi.end()));
  if (iter == data.end())
    std::cout << "The integer range does not contain the digits of pi.\n";
  else
  {
    std::cout << "Easy as pi: ";
    std::copy(iter, iter+pi.size(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';
  }
}
```

Binary Search Algorithms

The map container stores its elements in sorted order, so you can use any of the binary search algorithms, but map also has member functions that can take advantage of access to the internal structure of a map and, so, offer improved performance. Thus, the binary search algorithms are typically used on sequential containers, such as vector, when you know that they contain sorted data. If the input range is not properly sorted, the results are undefined: you might get the wrong answer; the program might crash; or something even worse might happen.

The binary_search function simply tests whether a sorted range contains a particular value. By default, values are compared using only the < operator. Another form of binary_search takes a comparison functor as an additional argument to perform the comparison.

WHAT'S IN A NAME?

The find function performs a linear search for a single item. The search function performs a linear search for a matching series of items. So why isn't binary_search called binary_find? On the other hand, find_end searches for the match that is furthest right in a range of values, so why isn't it called search_end? The equal function is completely different from equal_range, in spite of the similarity in their names.

The C++ standards committee did its best to apply uniform rules for algorithm names, such as appending _if to functions that take a functor argument but cannot be overloaded, but it faced some historical constraints with a number of names. What this means for you is that you have to keep a reference close at hand. Don't judge a function by its name, but read the description of what the function does and how it does it, before you decide whether it's the right function to use.

The lower_bound function is similar to binary_search, except it returns an iterator. The iterator points to the first occurrence of the value, or it points to a position where the value belongs if you want to insert the value into the range and keep the values in sorted order. The upper_bound function is similar to lower_bound, except it returns an iterator that points to the last position where you can insert the value and keep it in sorted order. If the value is found, that means upper_bound points to one position past the last occurrence of the value in the range. To put it another way, the range [lower_bound, upper_bound]) is the sub-range of every occurrence of the value in the sorted range. As with any range, if lower_bound == upper_bound, the result range is empty, which means the value is not in the search range.

Listing 43-7 shows a variation on Listing 43-1, sorting the integer vector and searching for a value using lower_bound to perform a binary search.

Listing 43-7. Searching for an Integer Using Binary Search

```
#include <algorithm>
#include <iostream>

#include "data.hpp"

int main()
{
  intvector data{};
  read_data(data);
  std::sort(data.begin(), data.end());
  write_data(data);
  intvec_iterator iter{std::lower_bound(data.begin(), data.end(), 42)};
  if (iter == data.end())
    std::cout << "Value 42 not found\n";
  else
  {
    *iter = 0;
    std::cout << "Value 42 changed to 0:\n";
    write_data(data);
  }
}
```

Only two lines changed: adding one extra line to sort the vector and changing find to lower_bound. To better understand how lower_bound and upper_bound really work, it helps to write a test program. The program reads some integers from the user into a vector, sorts the vector, and clears the I/O state bits on the standard input (std::cin.clear()), so you can enter some test values. The program then repeatedly asks for integers from the user and searches for each value using lower_bound and upper_bound. To help you understand exactly what these functions return, call the distance function to determine an iterator's position in a vector, as follows:

```
intvec_iterator iter{std::lower_bound(data.begin(), data.end(), 42)};
std::cout << "Index of 42 is " << std::distance(data.begin(), iter) << '\n';
```

The distance function (declared in <iterator>) takes an iterator range and returns the number of elements in the range. The return type is the iterator's difference_type, which is just an integer type, although the exact type (e.g., int or long int) depends on the implementation.

Write the test program. Then run the program with the following sample input:

```
9 4 2 1 5 4 3 6 2 7 4
```

What should the program print as the sorted vector?

____

Fill in Table 43-1 with the expected values for the lower and upper bounds of each value. Then run the program to check your answers.

Table 43-1. *Results of Testing Binary Search Functions*

Value	Expected Lower Bound	Expected Upper Bound	Actual Lower Bound	Actual Upper Bound
3				
4				
8				
0				
10				

Compare your test program with mine in Listing 43-8.

Listing 43-8. Exploring the `lower_bound` and `upper_bound` Functions

```
#include <algorithm>
#include <iostream>
#include <vector>

#include "data.hpp"

int main()
{
  intvector data{};
  read_data(data);
  std::sort(data.begin(), data.end());
  write_data(data);

  std::cin.clear();
  int test{};
  while (std::cin >> test)
  {
    intvec_iterator lb{std::lower_bound(data.begin(), data.end(), test)};
    intvec_iterator ub{std::upper_bound(data.begin(), data.end(), test)};
    std::cout << "lower bound = " << std::distance(data.begin(), lb) << '\n' <<
                 "upper bound = " << std::distance(data.begin(), ub) << '\n';
  }
}
```

When you want the lower bound and upper bound of the same range, call `equal_range`, which returns a `pair` of iterators. The `first` member of the `pair` is the lower bound, and the `second` is the upper bound.

Other useful linear functions include `count`, which takes an iterator range and value and returns the number of occurrences of the value in the range. Its counterpart `count_if` takes a predicate instead of a value and returns the number of times the predicate returns true.

Three more algorithms have a common pattern. They apply a predicate to every element in a range and return a `bool`.

- `all_of(first, last, predicate)` returns true if `predicate(element)` returns true for every element in the range [`first`, `last`].
- `any_of(first, last, predicate)` returns true if `predicate(element)` returns true for at least one element in the range [`first`, `last`].
- `none_of(first, last, predicate)` returns true if `predicate(element)` returns false for every element in the range [`first`, `last`].

You have already seen how min returns the minimum of two values. It has a partner, min_element, which takes an iterator range and returns the iterator for the minimum value in the range. Ditto for max_element. You can guess what minmax_element returns: a pair of iterators for the minimum and maximum values in the range. All three come in the usual overloaded forms: one uses the < operator, and the other takes an additional argument for a comparison predicate.

Unlike all the other algorithms, min, max, and minmax also have a form that takes a curly brace–enclosed list of values and returns the minimum, maximum, or both of those values. These are the only algorithms that take a single curly brace–enclosed list as a single argument. The only constraint is that all the values must have the same type. For example, the following

```
int a{1}, b{2}, c{3};
int smallest{ std::min({ a, b, c }) };
```

is equivalent to the traditional style of using iterators

```
int a{1}, b{2}, c{3};
std::vector<int> tmp{ a, b, c};
int smallest{ *std::min_element(tmp.begin(), tmp.end()) };
```

Comparing

To check whether two ranges are equal, that is, that they contain the same values, call the equal algorithm. This algorithm takes a start and one-past-the-end iterator for one range and the start of the second range. You must ensure that the two ranges have the same size. If every element of the two ranges is equal, it returns true. If any element doesn't match, it returns false. The function has two forms: pass only the iterators to equal, and it compares elements with the == operator; pass a comparison functor as the last argument, and equal compares elements by calling the functor. The first argument to the functor is the element from the first range, and the second argument is the element from the second range.

The mismatch function is the opposite. It compares two ranges and returns a std::pair of iterators that refer to the first elements that do not match. The first member in the pair is an iterator that refers to an element in the first range, and the second member refers to the second range. If the two ranges are equal, the return value is a pair of end iterators.

The lexicographical_compare algorithm sets the record for the longest algorithm name. It compares two ranges and determines whether the first range is "less than" the second. It does this by comparing the ranges one element at a time. If the ranges are equal, the function returns false. If the ranges are equal up to the end of one range, and the other range is longer, the shorter range is less than the longer range. If an element mismatch is found, whichever range contains the smaller element is the smaller range. All elements are compared using the < operator (or a caller-supplied predicate) and checked for equivalence, not equality. Recall that elements a and b are equivalent if the following is true:

```
not (a < b) and not (b < a)
```

If you apply lexicographical_compare to two strings, you get the expected less-than relationship, which explains the name. In other words, if you call this algorithm with the strings "hello" and "help", it returns true; if you call it with "help" and "hello", it returns false; and if you call it with "hel" and "hello", it returns true.

Write a test program that reads two sequences of integers into separate vectors. (Remember to clear the state after reading the first vector's data.) Then test the equal, mismatch, and lexicographical_compare functions on the two ranges. Remember that equal and mismatch require their input ranges to have the same size. You can ensure that you compare only the number of elements in the shorter vector by computing the end iterator instead of calling the end() member function.

```
std::equal(data1.begin(),
           data1.begin() + std::min(data1.size(), data2.size()),
           data2.begin())
```

Not all iterators allow addition, but a `vector`'s iterators do allow it. Adding an integer `n` to `begin()` offsets the iterator as though you had advanced it `n` times with the ++ operator. (Discover more about iterators in the next Exploration.)

Table 43-2 lists some suggested input data sets.

Table 43-2. *Suggested Data Sets for Testing Comparison Algorithms*

Data Set 1	Data Set 2
1 2 3 4 5	1 2 3
1 2 3	1 2 3 4 5
1 2 3 4 5	1 2 4 5
1 2 3	1 2 3

Compare your test program with mine in Listing 43-9.

Listing 43-9. Testing Various Comparison Algorithms

```
#include <algorithm>
#include <iostream>
#include <vector>

#include "data.hpp"

int main()
{
  intvector data1{};
  intvector data2{};

  read_data(data1);
  std::cin.clear();
  read_data(data2);

  std::cout << "data1: ";
  write_data(data1);
  std::cout << "data2: ";
  write_data(data2);

  auto data1_end(data1.begin() + std::min(data1.size(), data2.size()));

  std::cout << std::boolalpha;
  std::cout << "equal(data1, data2) = " <<
    equal(data1.begin(), data1_end, data2.begin()) << '\n';

  auto result(mismatch(data1.begin(), data1_end, data2.begin()));
  std::cout << "mismatch(data1, data2) = index " <<
   std::distance(data1.begin(), result.first) << '\n';

  std::cout << "lex_comp(data1, data2) = " <<
    std::lexicographical_compare(data1.begin(), data1.end(),
                                 data2.begin(), data2.end()) << '\n';
}
```

Rearranging Data

You've already seen the `sort` algorithm many times. Other algorithms are also adept at rearranging values in a range. The `merge` algorithm merges two sorted input ranges into a single output range. As always, you must ensure the output range has sufficient room to accept the entire merged result from both input ranges. The two input ranges can be different sizes, so `merge` takes five or six arguments: two for the first input range, two for the second input range, one for the start of the output range, and an optional argument for a functor to use instead of the < operator.

The `replace` algorithm scans an input range and replaces every occurrence of an old value with a new value. The replacement occurs in place, so you specify the range with the usual pair of iterators, but no write iterator. The `replace_if` function is similar but takes a predicate instead of an old value. **Write a program that reads a vector of integers and replaces all occurrences of values in the range [10, 20] with 0**. Reuse the `intrange` functor class or write a lambda. Compare your program with mine in Listing 43-10.

Listing 43-10. Using `replace_if` and `intrange` to Replace All Integers in [10, 20] with 0

```
#include <algorithm>

#include "data.hpp"
#include "intrange.hpp"

int main()
{
  intvector data{};
  read_data(data);
  write_data(data);
  std::replace_if(data.begin(), data.end(), intrange{10, 20}, 0);
  write_data(data);
}
```

Listing 43-11 shows the same program using a lambda.

Listing 43-11. Using `replace_if` and `intrange` to Replace All Integers in [10, 20] with 0

```
#include <algorithm>

#include "data.hpp"
#include "intrange.hpp"

int main()
{
  intvector data{};
  read_data(data);
  write_data(data);
  std::replace_if(data.begin(), data.end(),
    [](int x)
    {
      return x >= 10 and x <= 20;
    },
    0);
  write_data(data);
}
```

A fun algorithm is `random_shuffle`, which shuffles elements in place into random order. This function takes two arguments, specifying the range to shuffle. Another form of the function takes three arguments. The final argument is a functor that returns a random number in the range [0, `n`], where `n` is the size of the input range, or it can be a uniform random number generator from the standard library (similar to what `randomint` uses in Listing 43-5).

Use the *sequence.hpp* file (from Listing 42-6) and generate a vector of 100 sequential integers. Then shuffle it into random order and print it. Compare your solution with mine in Listing 43-12.

Listing 43-12. Shuffling Integers into Random Order

```
#include <algorithm>

#include "data.hpp"
#include "sequence.hpp"

int main()
{
  intvector data(100);
  std::generate(data.begin(), data.end(), sequence{1, 1});
  write_data(data);
  std::random_shuffle(data.begin(), data.end());
  write_data(data);
}
```

The `generate` algorithm repeatedly calls a functor with no arguments and copies the return value into an output range. It calls the functor once per element in the range, overwriting every element. The `generate_n` function takes an iterator for the start of a range and an integer for the size of the range. It then calls a functor (the third argument) once for each element of the range, copying the return value into the range. It is your responsibility to ensure that the range actually has that many elements in it. To use `generate_n` instead of `generate` in Listing 43-10, you could write

```
std::generate_n(data.begin(), data.size(), sequence(1, 1));
```

If you don't have to call a functor for every item of a range but instead want to fill a range with copies of the same value, call `fill`, passing a pair of iterators that specify a range, and a value. The value is copied into every element in the range. The `fill_n` function takes a starting iterator and an integer size to specify the target range.

The `transform` algorithm modifies items by calling a functor for each item in an input range. It writes the transformed results to an output range, which can be the same as the input range, resulting in modifying the range in place. You've seen this algorithm at work already, so I won't add much to what you already know. The function has two forms: unary and binary. The unary form takes one input range, the start of an output range, and a functor. It calls the functor for each element of the input range, copying the result to the output range. The output range can be the same as the input range, or it can be a separate range. As with all algorithms, you must ensure that the output range is large enough to store the results.

The binary form of transform takes an input range, the start of a second input range (it assumes the size is the same as the size of the first input range), the start of an output range, and a binary functor. The functor is called for each element in the input ranges; the first argument comes from the first input range, and the second argument comes from the second input range. As with the unary form, the function copies the result to the output range, which can be the same as either input range. Note that the types of the two input ranges do not have to be the same.

Copying Data

Some algorithms operate in place, and others copy their results to an output range. For example, reverse reverses items in place, and reverse_copy leaves the input range intact and copies the reversed items to an output range. If a copying form of an algorithm exists, its name has _copy appended. (Unless it is also a predicate form of a function, in which case it has _if appended after _copy, as in replace_copy_if.)

In addition to just plain copy, which you've seen many times already, the standard library offers copy_backward, which makes a copy but starts at the end and works toward the beginning, preserving the original order; copy_n, which takes the start of a range, a count, and a write iterator; and copy_if, which is like copy but takes a predicate and copies an element only if the predicate returns true. Distinguish copy_backward from reverse_copy. The latter starts at the beginning and works toward the end of the input range but copies the values into reverse order.

If you have to move elements instead of copy them, call std::move or std::move_backward. This std::move is different from the one you encountered in Exploration 39. This one is declared in <algorithm>. Like copy, the move algorithm takes two read iterators and a write iterator. It calls the other form of std::move for each element of the input range, moving the element into the output range.

As with all algorithms that write output, it is your responsibility to ensure the output range is large enough to handle everything you write to it. Some implementations of the standard library offer debugging modes to help detect violations of this rule. If your library offers such a feature, by all means, take full advantage of it.

Deleting Elements

The trickiest algorithms to use are those that "remove" elements. As you learned in Exploration 22, algorithms such as remove don't actually delete anything. Instead, they rearrange the elements in the range, so that all the elements slated for removal are packed at the end of the range. You can then decide either to use the sub-range of elements you want to keep or erase the "removed" elements by calling the erase member function.

The remove function takes an iterator range and a value, and it removes all elements equal to that value. You can also use a predicate with remove_if to remove all elements for which a predicate returns true. These two functions have copying counterparts that don't rearrange anything but merely copy the elements that are not being removed: remove_copy copies all the elements that are not equal to a certain value and remove_copy_if copies all elements for which a predicate returns false.

Another algorithm that removes elements is unique (and unique_copy). It takes an input range and removes all adjacent duplicates, thereby ensuring that every item in the range is unique. (If the range is sorted, then all duplicates are adjacent.) Both functions can take a comparison functor instead of using the default == operator.

Write a program that reads integers into a vector, erases all elements equal to zero, copies only those elements that lie in the range [24, 42] to another vector, sorts the other vector, and removes duplicates. Print the resulting vector. My solution is in Listing 43-13.

Listing 43-13. Erasing Elements from a Vector

```
#include <algorithm>
#include "data.hpp"
#include "intrange.hpp"
```

```
int main()
{
  intvector data{};
  read_data(data);
  data.erase(std::remove(data.begin(), data.end(), 0), data.end());
  intvector copy{};
  std::remove_copy_if(data.begin(), data.end(), std::back_inserter(copy),
                      [](int x) { return  x< 24 or x > 42; });
  std::sort(copy.begin(), copy.end());
  copy.erase(std::unique(copy.begin(), copy.end()), copy.end());
  write_data(copy);
}
```

Iterators

Algorithms and iterators are closely related. All the algorithms (except `min`, `max`, and `minmax`) take two or more iterators as arguments. To use algorithms effectively, you must understand iterators. Therefore, the next Exploration will help you master iterators, all five flavors. That's right. Iterators come in five different varieties. Keep reading to learn more.

EXPLORATION 44

Iterators

Iterators provide element-by-element access to a sequence of things. The things can be numbers, characters, or objects of almost any type. The standard containers, such as `vector`, provide iterator access to the container contents, and other standard iterators let you access input streams and output streams, for example. The standard algorithms require iterators for operating on sequences of things.

Until now, your view and use of iterators has been somewhat limited. Sure, you've used them, but do you really understand them? This Exploration helps you understand what's really going on with iterators.

Kinds of Iterators

So far, you have seen that iterators come in multiple varieties, in particular, read and write. The `copy` function, for example, takes two read iterators to specify an input range and one write iterator to specify the start of an output range. As always, specify the input range as a pair of read iterators: one that refers to the first element of the range and one that refers to the one-past-the-end element of the input range. The `copy` function returns a write iterator: the value of the result iterator after the copy is complete.

```
WriteIterator copy(ReadIterator start, ReadIterator end, WriteIterator result);
```

All this time, however, I've oversimplified the situation by referring to "read" and "write" iterators. In fact, C++ has five different categories of iterators: input, output, forward, bidirectional, and random access. Input and output iterators have the least functionality, and random access has the most. You can substitute an iterator with more functionality anywhere that calls for an iterator with less. Figure 44-1 illustrates the substitutability of iterators. Don't be misled by the figure, however. It does not show class inheritance. What makes an object an iterator is its behavior. If it fulfills all the requirements of an input iterator, for example, it is an input iterator, regardless of its type.

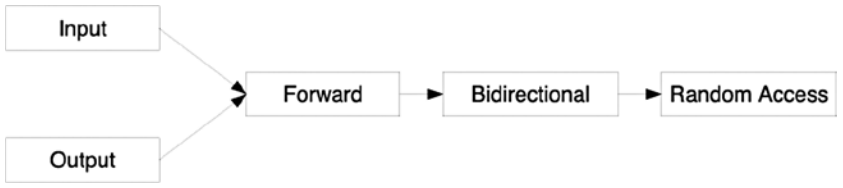


Figure 44-1. *Substitution tree for iterators*

All iterators can be copied and assigned freely. The result of a copy or an assignment is a new iterator that refers to the same item as the original iterator. The other characteristics depend on the iterator category, as described in the following sections.

Input Iterators

An input iterator, unsurprisingly, supports only input. You can read from the iterator (using the unary * operator) only once per iteration. You cannot modify the item that the iterator refers to. The ++ operator advances to the next input item. You can compare iterators for equality and inequality, but the only meaningful comparison is to compare an iterator with an end iterator. You cannot, in general, compare two input iterators to see if they refer to the same item.

That's about it. Input iterators are quite limited, but they are also extremely useful. Almost every standard algorithm expresses an input range in terms of two input iterators: the start of the input range and one past the end of the input range.

The `istream_iterator` type is an example of an input iterator. You can also treat any container's iterator as an input iterator; e.g., the iterator that a vector's `begin()` member function returns.

Output Iterators

An output iterator supports only output. You can assign to an iterator item (by applying the * operator to the iterator on the left-hand side of an assignment), but you cannot read from the iterator. You can modify the iterator value only once per iteration. The ++ operator advances to the next output item.

You *cannot* compare output iterators for equality or inequality.

In spite of the limitations on output iterators, they, too, are widely used by the standard algorithms. Every algorithm that copies data to an output range takes an output iterator to specify the start of the range.

One caution when dealing with output iterators is that you must ensure that wherever the iterator is actually writing has enough room to store the entire output. Any mistakes result in undefined behavior. Some implementations offer debugging iterators that can check for this kind of mistake, and you should certainly take advantage of such tools when they are available. Don't rely solely on debugging libraries, however. Careful code design, careful code implementation, and careful code review are absolutely necessary to ensure safety when using output (and other) iterators.

The `ostream_iterator` type is an example of an output iterator. You can also treat many container's iterators as output iterators; e.g., the iterator that a vector's `begin()` member function returns.

Forward Iterators

A forward iterator has all the functionality of an input iterator and an output iterator, and a little bit more. You can freely read from, and write to, an iterator item (still using the unary * operator), and you can do so as often as you wish. The ++ operator advances to the next item, and the `==` and `!=` operators can compare iterators to see if they refer to the same item or to the end position.

Some algorithms require forward iterators instead of input iterators. I glossed over that detail in the previous Explorations, because it rarely affects you. For example, the binary search algorithms require forward iterators to specify the input range, because they might have to refer to a particular item more than once. That means you cannot directly use an `istream_iterator` as an argument to, say, `lower_bound`, but then you aren't likely to try that in a real program. All the containers' iterators meet the requirements of forward iterators, so in practical terms, this restriction has little impact.

Bidirectional Iterators

A bidirectional iterator has all the functionality of a forward iterator, but it also supports the `--` operator, which moves the iterator backward one position to the previous item. As with any iterator, you are responsible for ensuring that you never advance the iterator past the end of the range or before the beginning.

The `reverse` and `reverse_copy` algorithms (and a few others) require bidirectional iterators. Most of the containers' iterators meet at least the requirements of bidirectional iterators, so you rarely have to worry about this restriction.

Random Access Iterators

A random access iterator is the most powerful iterator. It has all the functionality of all other iterators, plus you can move the iterator an arbitrary amount by adding or subtracting an integer.

You can subtract two iterators (provided they refer to the same sequence of objects) to obtain the distance between them. Recall from Exploration 43 that the `distance` function returns the distance between two iterators. If you pass forward or bidirectional iterators to the function, it advances the starting iterator one step at a time, until it reaches the end iterator. Only then will it know the distance. If you pass random access iterators, it merely subtracts the two iterators and immediately returns the distance between them.

You can compare random access iterators for equality or inequality. If the two iterators refer to the same sequence of objects, you can also use any of the relational operators. For random access iterators, `a < b` means `a` refers to an item earlier in the sequence than `b`.

Algorithms such as `sort` require random access iterators. The `vector` type provides random access iterators, but not all containers do. The `list` container, for example, implements a doubly linked list. It has only bidirectional iterators. Because you can't use the `sort` algorithm, the `list` container has its own `sort` member function. Learn more about `list` in Exploration 53.

Now that you know that vectors supply random access iterators, and you can compare random access iterators using relational operators, revisit Listing 10-4. Can you think of an easier way to write that program? (Hint: Consider a loop condition of `start < end`.) See my rewrite in Listing 44-1.

Listing 44-1. Comparing Iterators by Using the < Operator

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
  std::vector<int> data{};
  int x{};
  while (std::cin >> x)
    data.push_back(x);

  for (auto start(data.begin()), end(data.end()); start < end; ++start)
  {
    --end; // now end points to a real position, possibly start
    std::iter_swap(start, end); // swap contents of two iterators
  }

  std::copy(data.begin(), data.end(), std::ostream_iterator<int>(std::cout, "\n"));
}
```

So input, forward, bidirectional, and random access iterators all qualify as "read" iterators, and output, forward, bidirectional, and random access iterators all qualify as "write" iterators. An algorithm, such as `copy`, might require only input and output iterators. That is, the input range requires two input iterators. You can use any iterator that meets the requirements of an input iterator: input, forward, bidirectional, or random access. For the start of the output range, use any iterator that meets the requirements of an output iterator: output, forward, bidirectional, or random access.

Working with Iterators

The most common sources for iterators are the begin() and end() member functions that all containers (such as map and vector) provide. The begin() member function returns an iterator that refers to the first element of the container, and end() returns an iterator that refers to the position of the one-past-the-end element of the container.

What does begin() return for an empty container?

____

If the container is empty, begin() returns the same value as end(), that is, a special value that represents "past the end" and cannot be dereferenced. One way to test whether a container is empty is to test whether begin() == end(). (Even better, especially when you are writing a real program and not trying to illustrate the nature of iterators, is to call the empty() member function, which every container provides.)

The type of a container's iterator is always named iterator. The name is a nested member, so you refer to the iterator name by prefixing it with the container type.

```
std::map<std::string, int>::iterator map_iter;
std::vector<int>::iterator vector_iter;
```

Each container implements its iterator differently. All that matters to you is that the iterator fulfills the requirements of one of the standard categories.

The exact category of iterator depends on the container. A vector returns random access iterators. A map returns bidirectional iterators. Any library reference will tell you exactly what category of iterator each container supports.

A number of algorithms and container member functions also return iterators. For example, almost every function that performs a search returns an iterator that refers to the desired item. If the function cannot find the item, it returns the end iterator. The type of the return value is usually the same as the type of the iterators in the input range. Algorithms that copy elements to an output range return the result iterator.

Once you have an iterator, you can dereference it with * to obtain the value that it refers to (except for an output iterator, which you dereference only to assign a new value, and the end iterators, which you can never dereference). If the iterator refers to an object, and you want to access a member of the object, you can use the shorthand -> notation.

```
std::vector<std::string> lines(2, "hello");
std::string first{*lines.begin()};          // dereference the first item
std::size_t size{lines.begin()->size()};    // dereference and call a member function
```

You can advance an iterator to a new position by calling the next or advance function (declared in <iterator>). The advance function modifies the iterator that you pass as the first argument. The next function takes the iterator by value and returns a new iterator value. The second argument is the integer distance to advance the iterator. The second argument is optional for next; default is one. If the iterator is random access, the function adds the distance to the iterator. Any other kind of iterator must apply its increment (++) operator multiple times to advance the desired distance. For example:

```
std::vector<int> data{ 1, 2, 3, 4, 5 };
std::vector<int>::iterator iter{ data.begin() };
assert(4 == *std::next(iter, 3));
```

For a vector, std::next() is just like addition, but for other containers, such as std::map, it applies the increment operator multiple times to reach the desired destination. If you have a reverse iterator, you can pass a negative distance or call std::prev(). If the iterator is bidirectional, the second argument can be negative to go backward. You can advance an input iterator but not an output iterator. Reusing the sequence functor from Exploration 42, read the program in Listing 44-2.

Listing 44-2. Advancing an Iterator

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

#include "data.hpp"       // see Listing 43-2.

#include "sequence.hpp"   // see Listing 42-6.

int main()
{
  intvector data(10);
  std::generate(data.begin(), data.end(), sequence{0, 2}); // fill with even numbers
  intvec_iterator iter{data.begin()};
  std::advance(iter, 4);
  std::cout << *iter << ", ";
  iter = std::prev(iter, 2);
  std::cout << *iter << '\n';
}
```

What does the program print?____

The data vector is filled with even numbers, starting at 0. The iterator, iter, initially refers to the first element of the vector, namely, 0. The iterator advances four positions, to value 8, and then back two positions, to 4. So the output is

```
8, 4
```

Declaring variables to store iterators is clumsy. The type names are long and cumbersome. Therefore, I often use auto to define a variable. If I really need to name a type, say, because I have to write a function that takes an iterator parameter, I use typedef declarations to create shorter aliases for the long, clumsy type names, as follows:

```
typedef std::vector<std::string>::iterator strvec_iterator;

std::vector<std::string> lines(2, "hello");
strvec_iterator iter{lines.begin()};
std::string first{*iter};          // dereference the first item
std::size_t size{iter->size()};    // dereference and call a member function
```

const_iterator vs. const iterator

A minor source of confusion is the difference between a const_iterator and const iterator. An output iterator (and any iterator that also meets the requirements of an output iterator, namely, forward, bidirectional, and random access) lets you modify the item it references. For some forward iterators (and bidirectional and random access), you want to treat the data in the range as read-only. Even though the iterator itself meets the requirements of a forward iterator, your immediate need might be only for an input iterator.

You might think that declaring the iterator const would help. After all, that's how you ask the compiler to help you, by preventing accidental modification of a variable: declare the variable with the const specifier. What do you think? **Will it work?** ________________

If you aren't sure, try a test. Read Listing 44-3 and **predict its output**.

Listing 44-3. Printing the Middle Item of a Series of Integers

```
#include <iostream>
#include "data.hpp"

int main()
{
  intvector data{};
  read_data(data);
  const intvec_iterator iter{data.begin()};
  std::advance(iter, data.size() / 2); // move to middle of vector
  if (not data.empty())
    std::cout << "middle item = " << *iter << '\n';
}
```

Can you see why the compiler refuses to compile the program? Maybe you can't see the precise reason, buried in the compiler's error output. (The next section will discuss this problem at greater length.) The error is that the variable `iter` is `const`. You cannot modify the iterator, so you cannot advance it to the middle of the vector.

Instead of declaring the iterator itself as `const`, you have to tell the compiler that you want the iterator to refer to `const` data. If the vector itself were `const`, the `begin()` function would return exactly such an iterator. You could freely modify the iterator's position, but you could not modify the value that the iterator references. The name of the iterator that this function returns is `const_iterator` (with underscore).

In other words, every container actually has two different `begin()` functions. One is a `const` member function and returns `const_iterator`. The other is not a `const` member function; it returns a plain `iterator`. As with any `const` or non-`const` member function, the compiler chooses one or the other, depending on whether the container itself is `const`. If the container is not `const`, you get the non-`const` version of `begin()`, which returns a plain `iterator`, and you can modify the container contents through the iterator. If the container is `const`, you get the `const` version of `begin()`, which returns a `const_iterator`, which prevents you from modifying the container's contents. You can also force the issue by calling `cbegin()`, which always returns `const_iterator`, even for a non-`const` object.

Rewrite Listing 44-3 to use a `const_iterator`. Your program should look something like Listing 44-4.

Listing 44-4. Really Printing the Middle Item of a Series of Integers

```
#include <iostream>

#include "data.hpp"

int main()
{
  intvector data{};
  read_data(data);
  intvector::const_iterator iter{data.begin()};
  std::advance(iter, data.size() / 2); // move to middle of vector
  if (not data.empty())
    std::cout << "middle item = " << *iter << '\n';
}
```

Prove to yourself that you cannot modify the data when you have a `const_iterator`. **Make a further modification to your program to negate the middle value**. Now your program should look like Listing 44-5.

Listing 44-5. Negating the Middle Value in a Series of Integers

```
#include <iostream>

#include "data.hpp"

int main()
{
  intvector data{};
  read_data(data);
  intvector::const_iterator iter{data.begin()};
  std::advance(iter, data.size() / 2); // move to middle of vector
  if (not data.empty())
    *iter = -*iter;
  write_data(data);
}
```

If you change `const_iterator` to `iterator`, the program works. **Do it**.

Error Messages

When you compiled Listing 44-3, the compiler issued an error message, or *diagnostic,* as the C++ standard writers call it. For example, the compiler that I use every day, g++, prints the following:

```
In file included from /usr/include/c++/4.8/bits/stl_algobase.h:66:0,
                 from /usr/include/c++/4.8/bits/char_traits.h:39,
                 from /usr/include/c++/4.8/ios:40,
                 from /usr/include/c++/4.8/ostream:38,
                 from /usr/include/c++/4.8/iostream:39,
                 from list4403.cpp:1:
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h: In instantiation of 'void↩
std::__advance(_RandomAccessIterator&, _Distance, std::random_access_iterator_tag) [with↩
_RandomAccessIterator = const __gnu_cxx::__normal_iterator<int*, std::vector<int> >;
_Distance = long int]':
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h:177:61:   required from 'void↩
std::advance(_InputIterator&, _Distance) [with _InputIterator = const __gnu_cxx::__normal_iterator<int*,↩
std::vector<int> >; _Distance = long unsigned int]'
list4403.cpp:9:37:   required from here
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h:156:11: error: no match for 'operator+=' (operand↩
types are 'const __gnu_cxx::__normal_iterator<int*, std::vector<int> >' and 'long int')
       __i += __n;
           ^
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h:156:11: note: candidate is:
In file included from /usr/include/c++/4.8/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/4.8/bits/char_traits.h:39,
                 from /usr/include/c++/4.8/ios:40,
                 from /usr/include/c++/4.8/ostream:38,
                 from /usr/include/c++/4.8/iostream:39,
                 from list4403.cpp:2:
```

```
/usr/include/c++/4.8/bits/stl_iterator.h:774:7: note: __gnu_cxx::__normal_iterator<_Iterator, _Container>&↩
__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator+=(const difference_type&) [with _Iterator↩
= int*; _Container = std::vector<int>; __gnu_cxx::__normal_iterator<_Iterator,↩
_Container>::difference_type = long int] <near match>
       operator+=(const difference_type& __n)
       ^
/usr/include/c++/4.8/bits/stl_iterator.h:774:7: note:   no known conversion for implicit 'this'
parameter from↩
'const __gnu_cxx::__normal_iterator<int*, std::vector<int> >*' to '__gnu_cxx::__normal_iterator<int*,↩
std::vector<int> >*'
```

So what does all that gobbledygook mean? Although a C++ expert can figure it out, it may not be much help to you. Buried in the middle is the line number and source file that identify the source of the error. That's where you have to start looking. The compiler didn't detect the error until it started working through various `#include` files. These file names depend on the implementation of the standard library, so you can't always tell from those file names what is the actual error.

In this case, the error arises from within the `std::advance` function. That's when the compiler detects that it has a `const` iterator, but it does not have any functions that work with a `const` iterator. Instead of complaining about the `const`-ness, however, all it manages to do is to complain that it lacks a "match" for the function it seeks. That means it is looking for argument types that match the parameter types, to resolve an overloaded operator. Because a `const` argument cannot match a non-`const` parameter, the compiler failed to find an overload that matches the arguments.

Don't give up hope for ever understanding C++ compiler error messages. By the end of the book, you will have gained quite a bit more knowledge that will help you understand how the compiler and library really work, and that understanding will help you make sense of these error messages. If you can, compile your code with clang++, which does a good job of issuing helpful messages. Even if your production code must be compiled with a different compiler, you can use clang++ to help diagnose problems.

My advice for dealing with the deluge of confusing error messages is to start by finding the first mention of your source file. That should tell you the line number that gives rise to the problem. Check the source file. You might see an obvious mistake. If not, check the error message text. Ignore the "instantiated from here" and similar messages. Try to find the real error message, which often starts with `error:` instead of `warning:` or `note:`.

Specialized Iterators

The `<iterator>` header defines a number of useful, specialized iterators, such as `back_inserter`, which you've seen several times already. Strictly speaking, `back_inserter` is a function that returns an iterator, but you rarely have to know the exact iterator type.

In addition to `back_inserter`, you can also use `front_inserter`, which also takes a container as an argument and returns an output iterator. Every time you assign a value to the dereferenced iterator, it calls the container's `push_front` member function to insert the value at the start of the container.

The `inserter` function takes a container and an iterator as arguments. It returns an output iterator that calls the container's `insert` function. The `insert` member function requires an `iterator` argument, specifying the position at which to insert the value. The `inserter` iterator initially passes its second argument as the insertion position. After each insertion, it updates its internal iterator, so subsequent insertions go into subsequent positions. In other words, `inserter` just does the right thing.

Other specialized iterators include `istream_iterator` and `ostream_iterator`, which you've also seen. An `istream_iterator` is an input iterator that extracts values from a stream when you dereference the iterator. With no arguments, the `istream_iterator` constructor creates an end-of-stream iterator. An iterator is equal to the end-of-stream iterator when an input operation fails.

An `ostream_iterator` is an output iterator. The constructor takes an output stream and an optional string as arguments. Assigning to the dereferenced iterator writes a value to the output stream, optionally followed by the string (from the constructor).

Another specialized iterator is the`reverse_iterator` class. It adapts an existing iterator (called the *base* iterator), which must be bidirectional (or random access). When the reverse iterator goes forward (++), the base iterator goes backward (--). Containers that support bidirectional iterators have `rbegin()` and `rend()` member functions, which return reverse iterators. The `rbegin()` function returns a reverse iterator that points to the last element of the container, and `rend()` returns a special reverse iterator value that represents one position before the beginning of the container. Thus, you treat the range [`rbegin()`, `rend()`] as a normal iterator range, expressing the values of the container in reverse order.

C++ doesn't permit an iterator to point to one position before the beginning, so reverse iterators have a somewhat funky implementation. Ordinarily, implementation details don't matter, but `reverse_iterator` exposes this particular detail in its `base()` member function, which returns the base iterator.

I could tell you what the base iterator actually is, but that would deprive you of the fun. **Write a program to reveal the nature of the `reverse_iterator`'s base iterator**. (Hint: Fill a vector with a sequence of integers. Use a reverse iterator to get to the middle value. Compare with the value of the iterator's `base()` iterator.)

If a `reverse_iterator` points to position x of a container, what does its `base()` iterator point to?

____

If you did not answer *x* + 1, try running the program in Listing 44-6.

Listing 44-6. Revealing the Implementation of `reverse_iterator`

```
#include <algorithm>
#include <cassert>
#include <iostream>

#include "data.hpp"
#include "sequence.hpp"

int main()
{
  intvector data(10);
  std::generate(data.begin(), data.end(), sequence(1));
  write_data(data);                             // prints { 1 2 3 4 5 6 7 8 9 10 }
  intvector::iterator iter{data.begin()};
  iter = iter + 5;                              // iter is random access
  std::cout << *iter << '\n';                   // prints 5

  intvector::reverse_iterator rev{data.rbegin()};
  std::cout << *rev << '\n';                    // prints 10
  rev = rev + 4;                                // rev is also random access
  std::cout << *rev << '\n';                    // prints 6
  std::cout << *rev.base() << '\n';             // prints 7
  std::cout << *data.rend().base() << '\n';     // prints 0
  assert(data.rbegin().base() == data.end());
  assert(data.rend().base()   == data.begin());
}
```

Now do you see? The base iterator always points to one position *after* the reverse iterator's position. That's the trick that allows `rend()` to point to a position "before the beginning," even though that's not allowed. Under the hood, the `rend()` iterator actually has a base iterator that points at the first item in the container, and the `reverse_iterator`'s implementation of the `*` operator performs the magic of taking the base iterator, retreating one position, and then dereferencing the base iterator.

As you can see, iterators are a little more complicated than they initially seem to be. Once you understand how they work, however, you will see that they are actually quite simple, powerful, and easy-to-use. But first, it's time to pick up a few more important C++ programming techniques. The next Exploration introduces exceptions and exception-handling, necessary topics for properly handling programmer and user errors.

EXPLORATION 45

Exceptions

You may have been dismayed by the lack of error checking and error handling in the Explorations so far. That's about to change. C++, like most modern programming languages, supports exceptions as a way to jump out of the normal flow of control in response to an error or other exceptional condition. This Exploration introduces exceptions: how to throw them, how to catch them, when the language and library use them, and when and how you should use them.

Introducing Exceptions

Exploration 9 introduced `vector`'s `at` member function, which retrieves a vector element at a particular index. At the time, I wrote that most programs you read would use square brackets instead. Now is a good time to examine the difference between square brackets and the `at` function. First, take a look at two programs. Listing 45-1 shows a simple program that uses a vector.

Listing 45-1. Accessing an Element of a Vector

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> data{ 10, 20 };
  data.at(5) = 0;
  std::cout << data.at(5) << '\n';
}
```

What do you expect to happen when you run this program?

____

Try it. **What actually happens?**

____

The vector index, 5, is out of bounds. The only valid indices for `data` are 0 and 1, so it's no wonder that the program terminates with a nastygram. Now consider the program in Listing 45-2.

Listing 45-2. A Bad Way to Access an Element of a Vector

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> data{ 10, 20 };
  data[5] = 0;
  std::cout << data[5] << '\n';
}
```

What do you expect to happen when you run this program?

____

Try it. **What actually happens?**

____

The vector index, 5, is still out of bounds. If you still receive a nastygram, you get a different one than before. On the other hand, the program might run to completion without indicating any error. You might find that disturbing, but such is the case of undefined behavior. Anything can happen.

That, in a nutshell, is the difference between using subscripts ([]) and the at member function. If the index is invalid, the at member function causes the program to terminate in a predictable, controlled fashion. You can write additional code and avoid termination, take appropriate actions to clean up prior to termination, or let the program end.

The subscript operator, on the other hand, results in undefined behavior if the index is invalid. Anything can happen, so you have no control—none whatsoever. If the software is controlling, say, an airplane, then "anything" involves many options that are too unpleasant to imagine. On a typical desktop workstation, a more likely scenario is that the program crashes, which is a good thing, because it tells you that something went wrong. The worst possible consequence is that nothing obvious happens, and the program silently uses a garbage value and keeps running.

The at member function, and many other functions, can *throw exceptions* to signal an error. When a program throws an exception, the normal, statement-by-statement progression of the program is interrupted. Instead, a special exception-handling system takes control of the program. The standard gives some leeway in how this system actually works, but you can imagine that it forces functions to end and destroys local objects and parameters, although the functions do not return a value to the caller. Instead, functions are forcefully ended, one at a time, and a special code block *catches* the exception. Use the try-catch statement to set up these special code blocks in a program. A catch block is also called an *exception handler*. Normal code execution resumes after the handler finishes its work:

```
try {
  throw std::runtime_error("oops");
} catch (std::runtime_error const& ex) {
  std::cerr << ex.what() << '\n';
}
```

When a program throws an exception (with the throw keyword), it throws a value, called an exception object, which can be an object of nearly any type. By convention, exception types, such as std::runtime_error, inherit from the std::exception class or one of several subclasses that the standard library provides. Third-party class libraries sometimes introduce their own exception base class.

An exception handler also has an object declaration, which has a type, and the handler accepts only exception objects of the same type or of a derived type. If no exception handler has a matching type, or if you don't write any handler at all, the program terminates, as happens with Listing 45-1. The remainder of this Exploration examines each aspect of exception handling in detail.

Catching Exceptions

An exception handler is said to *catch* an exception. Write an exception handler at the end of a `try`: the `try` keyword is followed by a compound statement (it must be compound), followed by a series of *handlers*. Each handler starts with a `catch` keyword, followed by parentheses that enclose the declaration of an exception-handler object. After the parentheses is a compound statement that is the body of the exception handler.

When the type of the exception object matches the type of the exception-handler object, the handler is deemed a match, and the handler object is initialized with the exception object. The handler declaration is usually a reference, which avoids copying the exception object unnecessarily. Most handlers don't have to modify the exception object, so the handler declaration is typically a reference to `const`. A "match" is when the exception object's type is the same as the handler's declared type or a class derived from the handler's declared type, ignoring whether the handler is `const` or a reference.

The exception-handling system destroys all objects that it constructed in the `try` part of the statement prior to throwing the exception, then it transfers control to the handler, so the handler's body runs normally, and control resumes with the statement after the end of the entire `try-catch` statement, that is, after the statement's last `catch` handler. The handler types are tried in order, and the first match wins. Thus, you should always list the most specific types first and base class types later.

A base class exception handler type matches any exception object of a derived type. To handle all exceptions that the standard library might throw, write the handler to catch `std::exception` (declared in `<exception>`), which is the base class for all standard exceptions. Listing 45-3 demonstrates some of the exceptions that the `std::string` class can throw. Try out the program by typing strings of varying length.

Listing 45-3. Forcing a `string` to Throw Exceptions

```
#include <cstdlib>
#include <exception>
#include <iostream>
#include <stdexcept>
#include <string>

int main()
{
  std::string line{};
  while (std::getline(std::cin, line))
  {
    try
    {
      line.at(10) = ' ';                              // can throw out_of_range
      if (line.size() < 20)
         line.append(line.max_size(), '*'); // can throw length_error
      for (std::string::size_type size(line.size());
           size < line.max_size();
           size = size * 2)
      {
        line.resize(size);                            // can throw bad_alloc
      }
      line.resize(line.max_size());                   // can throw bad_alloc
      std::cout << "okay\n";
    }
    catch (std::out_of_range const& ex)
    {
       std::cout << ex.what() << '\n';
       std::cout << "string index (10) out of range.\n";
    }
    catch (std::length_error const& ex)
    {
      std::cout << ex.what() << '\n';
      std::cout << "maximum string length (" << line.max_size() << ") exceeded.\n";
    }
    catch (std::exception const& ex)
    {
      std::cout << "other exception: " << ex.what() << '\n';
    }
    catch (...)
    {
      std::cout << "Unknown exception type. Program terminating.\n";
      std::abort();
    }
  }
}
```

If you type a line that contains 10 or fewer characters, the `line.at(10)` expression throws a `std::out_of_range` exception. If the string has more than 10 characters, but fewer than 20, the program tries to append the maximum string size repetitions of an asterisk (`'*'`), which results in `std::length_error`. If the initial string is longer than 20 characters, the program tries to increase the string size, using ever-growing sizes. Most likely, the size will eventually exceed available memory, in which case the `resize()` function will throw `std::bad_alloc`. If you have lots and lots of memory, the next error situation forces the string size to the limit that `string` supports and then tries to add another character to the string, which causes the `push_back` function to throw `std::length_error`. (The `max_size` member function returns the maximum number of elements that a container, such as `std::string`, can contain.)

The base class handler catches any exceptions that the first two handlers miss; in particular, it catches `std::bad_alloc`. The `what()` member function returns a string that describes the exception. The exact contents of the string vary by implementation. Any nontrivial application should define its own exception classes and hide the standard library exceptions from the user. In particular, the strings returned from `what()` are implementation-defined and are not necessarily helpful. Catching `bad_alloc` is especially tricky, because if the system is running out of memory, the application might not have enough memory to save its data prior to shutting down. You should always handle `bad_alloc` explicitly, but I wanted to demonstrate a handler for a base class.

The final `catch` handler uses an ellipsis (`...`) instead of a declaration. This is a catch-all handler that matches any exception. If you use it, it must be last, because it matches every exception object, of any type. Because the handler doesn't know the type of the exception, it has no way to access the exception object. This catch-all handler prints a message and then calls `std::abort()` (declared in `<exception>`), which immediately ends the program. Because the `std::exception` handler catches all standard library exceptions, the final catch-all handler is not really needed, but I wanted to show you how it works.

Throwing Exceptions

A *throw expression* throws an exception. The expression consists of the `throw` keyword followed by an expression, namely, the exception object. The standard exceptions all take a `string` argument, which becomes the value returned from the `what()` member function.

```
throw std::out_of_range("index out of range");
```

The messages that the standard library uses for its own exceptions are implementation-defined, so you cannot rely on them to provide any useful information.

You can throw an exception anywhere an expression can be used, sort of. The type of a throw expression is `void`, which means it has no type. Type `void` is not allowed as an operand for any arithmetic, comparison, etc., operator. Thus, realistically, a `throw` expression is typically used in an expression statement, all by itself.

You can throw an exception inside a catch handler, which low-level code and libraries often do. You can throw the same exception object or a brand-new exception. To re-throw the same object, use the `throw` keyword without any expression.

```
catch (std::out_of_range const& ex)
{
  std::cout << "index out of range\n";
  throw;
}
```

A common case for re-throwing an exception is inside a catch-all handler. The catch-all handler performs some important cleanup work and then propagates the exception so the program can handle it.

If you throw a new exception, the exception-handling system takes over normally. Control leaves the `try-catch` block immediately, so the same handler cannot catch the new exception.

Program Stack

To understand what happens when a program throws an exception, you must first understand the nature of the *program stack*, sometimes called the *execution stack*. Procedural and similar languages use a stack at runtime to keep track of function calls, function arguments, and local variables. The C++ stack also helps keep track of exception handlers.

When a program calls a function, the program pushes a *frame* onto the stack. The frame has information such as the instruction pointer and other registers, arguments to the function, and possibly some memory for the function's return value. When a function starts, it might set aside some memory on the stack for local variables. Each local scope pushes a new frame onto the stack. (The compiler might be able to optimize away a physical frame for some local scopes, or even an entire function. Conceptually, however, the following applies.)

While a function executes, it typically constructs a variety of objects: function arguments, local variables, temporary objects, and so on. The compiler keeps track of all the objects the function must create, so it can properly destroy them when the function returns. Local objects are destroyed in the opposite order of their creation.

Frames are dynamic, that is, they represent function calls and the flow of control in a program, not the static representation of source code. Thus, a function can call itself, and each call results in a new frame on the stack, and each frame has its own copy of all the function arguments and local variables.

When a program throws an exception, the normal flow of control stops, and the C++ exception-handling mechanism takes over. The exception object is copied to a safe place, off the execution stack. The exception-handling code looks through the stack for a `try` statement. When it finds a `try` statement, it checks the types for each handler in turn, looking for a match. If it doesn't find a match, it looks for the next `try` statement, farther back in the stack. It keeps looking until it finds a matching handler or it runs out of frames to search.

When it finds a match, it pops frames off the execution stack, calling destructors for all local objects in each popped frame, and continues to pop frames until it reaches the handler. Popping frames from the stack is also called *unwinding* the stack.

After unwinding the stack, the exception object initializes the handler's exception object, and then the `catch` body is executed. After the `catch` body exits normally, the exception object is freed, and execution continues with the statement that follows the end of the last sibling `catch` block.

If the handler throws an exception, the search for a matching handler starts anew. A handler cannot handle an exception that it throws, nor can any of its sibling handlers in the same `try` statement.

If no handler matches the exception object's type, the `std::terminate` function is called, which aborts the program. Some implementations will pop the stack and free local objects prior to calling `terminate`, but others won't.

Listing 45-4 can help you visualize what is going on inside a program when it throws and catches an exception.

Listing 45-4. Visualizing an Exception

```
#include <exception>
#include <iostream>
#include <string>

/// Make visual the construction and destruction of objects.
class visual
{
public:
  visual(std::string const& what)
  : id_{serial_}, what_{what}
  {
    ++serial_;
    print("");
  }
  visual(visual const& ex)
  : id_{ex.id_}, what_{ex.what_}
  {
    print("copy ");
  }
  ~visual()
  {
    print("~");
  }
  void print(std::string const& label)
  const
  {
    std::cout << label << "visual(" << what_ << ": " << id_ << ")\n";
  }
private:
  static int serial_;
  int const id_;
  std::string const what_;
};

int visual::serial_{0};

void count_down(int n)
{
  std::cout << "start count_down(" << n << ")\n";
  visual v{"count_down local"};
  try
  {
    if (n == 3)
      throw visual("exception");
    else if (n > 0)
      count_down(n - 1);
  }
  catch (visual ex)
49   {
50     ex.print("catch ");
51     throw;
52   }
53   std::cout << "end count_down(" << n << ")\n";
54 }
55
56 int main()
57 {
58   try
59   {
60     count_down(2);
61     count_down(4);
62   }
63   catch (visual const ex)
64   {
65     ex.print("catch ");
66   }
67   std::cout << "All done!\n";
68 }
```

The `visual` class helps show when and how objects are constructed, copied, and destroyed. The `count_down` function throws an exception when its argument equals 3, and it calls itself when its argument is positive. The recursion stops for non-positive arguments. To help you see function calls, it prints the argument upon entry to, and exit from, the function.

The first call to `count_down` does not trigger the exception, so you should see normal creation and destruction of the local `visual` object. **Write exactly what the program should print as a result of line 60 (`count_down(2)`).**

The next call to `count_down` from `main` (line 61) allows `count_down` to recurse once before throwing an exception. So `count_down(4)` calls `count_down(3)`. The local object, `v`, is constructed inside the frame for `count_down(4)`, and a

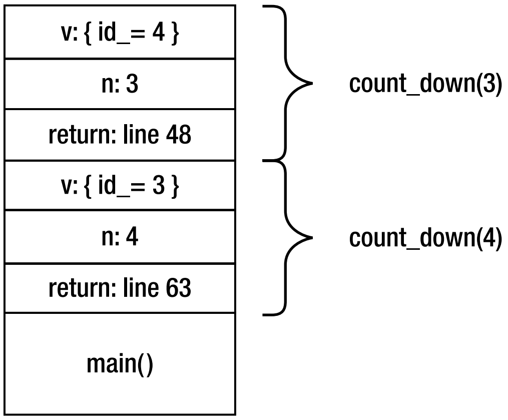


Figure 45-1. *Program stack when the exception is thrown*

new instance of v is constructed inside the frame for `count_down(3)`. Then the exception object is created and thrown. (See Figure 45-1.)

The exception is caught inside `count_down`, so its frame is not popped. The exception object is then copied to `ex` (line 48), and the exception handler begins. It prints a message and then re-throws the original exception object (line 51). The exception-handling mechanism treats this exception the same way it treats any other: the `try` statement's frame is popped, and then the `count_down` function's frame is popped. Local objects are destroyed (including `ex` and `v`). The final statement in `count_down` does not execute.

The stack is unwound, and the `try` statement inside the call to `count_down(4)` is found, and once again, the exception object is copied to a new instance of `ex`. (See Figure 45-2.) The exception handler prints a message and re-throws the original exception. The `count_down(4)` frame is popped, returning control to the `try` statement in `main`. Again, the final statement in `count_down` does not execute.

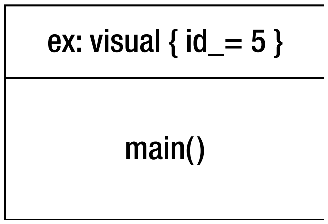


Figure 45-2. *Program stack after re-throwing exception*

The exception handler in `main` gets its turn, and this handler prints the exception object one last time (line 63). After the handler prints a message, and the `catch` body reaches its end, the local exception object and the original exception object are destroyed. Execution then continues normally on line 67. The final output is

```
start count_down(2)
visual(count_down local: 0)
start count_down(1)
visual(count_down local: 1)
start count_down(0)
visual(count_down local: 2)
end count_down(0)
~visual(count_down local: 2)
end count_down(1)
~visual(count_down local: 1)
end count_down(2)
~visual(count_down local: 0)
start count_down(4)
visual(count_down local: 3)
start count_down(3)
visual(count_down local: 4)
visual(exception: 5)
copy visual(exception: 5)
catch visual(exception: 5)
~visual(exception: 5)
~visual(count_down local: 4)
copy visual(exception: 5)
catch visual(exception: 5)
~visual(exception: 5)
~visual(count_down local: 3)
copy visual(exception: 5)
catch visual(exception: 5)
~visual(exception: 5)
~visual(exception: 5)
All done!
```

Standard Exceptions

The standard library defines several standard exception types. The base class, `exception`, is declared in the `<exception>` header. Most of the other exception classes are defined in the `<stdexcept>` header. If you want to define your own exception class, I recommend deriving it from one of the standard exceptions in `<stdexcept>`.

The standard exceptions are divided into two categories (with two base classes that derive directly from `exception`).

- Runtime errors (`std::runtime_error`) are exceptions that you cannot detect or prevent merely by examining the source code. They arise from conditions that you can anticipate, but not prevent.
- Logic errors (`std::logic_error`) are the result of programmer error. They represent violations of preconditions, invalid function arguments, and other errors that the programmer should prevent in code.

The other standard exception classes in `<stdexcept>` derive from these two. Most standard library exceptions are logic errors. For example, `out_of_range` inherits from `logic_error`. The `at` member function (and others) throws `out_of_range` when the index is out of range. After all, you should check indices and sizes, to be sure your vector and string usage are correct, and not rely on exceptions. The exceptions are there to provide clean, orderly shutdown of your program when you do make a mistake (and we all make mistakes).

Your library reference tells you which functions throw which exceptions, such as `at` can throw `out_of_range`. Many functions might throw other, undocumented exceptions too, depending on the library's and compiler's implementation. In general, however, the standard library uses few exceptions. Instead, most of the library yields undefined behavior when you provide bad input. The I/O streams do not ordinarily throw any exceptions, but you can arrange for them to throw exceptions when bad errors happen, as I explain in the next section.

I/O Exceptions

You learned about I/O stream state bits in Exploration 31. State bits are important, but checking them repeatedly is cumbersome. In particular, many programs fail to check the state bits of output streams, especially when writing to the standard output. That's just plain, old-fashioned laziness. Fortunately, C++ offers an avenue for programmers to gain I/O safety without much extra work: the stream can throw an exception when I/O fails.

In addition to state bits, each stream also has an *exception mask*. The exception mask tells the stream to throw an exception if a corresponding state bit changes value. For example, you could set `badbit` in the exception mask and never write an explicit check for this unlikely occurrence. If a serious I/O error were to occur, causing `badbit` to become set, the stream would throw an exception. You can write a handler at a high level to catch the exception and terminate the program cleanly, as shown in Listing 45-5.

Listing 45-5. Using an I/O Stream Exception Mask

```
#include <iostream>

int main()
{
  std::cin.exceptions(std::ios_base::badbit);
  std::cout.exceptions(std::ios_base::badbit);

  int x{};
  try
  {
    while (std::cin >> x)
      std::cout << x << '\n';
    if (not std::cin.eof()) // failure without eof means invalid input
      std::cerr << "Invalid integer input. Program terminated.\n";
  }
  catch(std::ios_base::failure const& ex)
  {
    std::cerr << "Major I/O failure! Program terminated.\n";
    std::terminate();
  }
}
```

As you can see, the exception class is named `std::ios_base::failure`. Also note a new output stream: `std::cerr`. The `<iostream>` header actually declares several standard I/O streams. So far, I've used only `cin` and `cout`, because that's all we've needed. The `cerr` stream is an output stream dedicated to error output. In this case, separating normal output (to `cout`) from error output (to `cerr`) is important, because `cout` might have a fatal error (say, a disk is full), so any attempt to write an error message to `cout` would be futile. Instead, the program writes the message to `cerr`. There's no guarantee that writing to `cerr` would work, but at least there's a chance; for example, the user might redirect the standard output to a file (and thereby risk encountering a disk-full error), while allowing the error output to appear on a console.

Recall that when an input stream reaches the end of the input, it sets eofbit in its state mask. Although you can also set this bit in the exceptions mask, I can't see any reason why you would want to. If an input operation does not read anything useful from the stream, the stream sets failbit. The most common reasons that the stream might not read anything is end of file (eofbit is set) or an input formatting error (e.g., text in the input stream when the program tries to read a number). Again, it's possible to set failbit in the exception mask, but most programs rely on ordinary program logic to test the state of an input stream. Exceptions are for exceptional conditions, and end-of-file is a normal occurrence when reading from a stream.

The loop ends when failbit is set, but you have to test further to discover whether failbit is set, because of a normal end-of-file condition or because of malformed input. If eofbit is also set, you know the stream is at its end. Otherwise, failbit must be owing to malformed input.

As you can see, exceptions are not the solution for every error situation. Thus, badbit is the only bit in the exception mask that makes sense for most programs, especially for input streams. An output stream sets failbit if it cannot write the entire value to the stream. Usually, such a failure occurs because of an I/O error that sets badbit, but it's at least theoretically possible for output failure to set failbit without also setting badbit. In most situations, any output failure is cause for alarm, so you might want to throw an exception for failbit with output streams and badbit with input streams.

```
std::cin.exceptions(std::ios_base::badbit);
std::cout.exceptions(std::ios_base::failbit);
```

Custom Exceptions

Exceptions simplify coding by removing exceptional conditions from the main flow of control. You can and should use exceptions for many error situations. For example, the rational class (most recently appearing in Exploration 40) has, so far, completely avoided the issue of division by zero. A better solution than invoking undefined behavior (which is what happens when you divide by zero) is to throw an exception anytime the denominator is zero. Define your own exception class by deriving from one of the standard exception base classes, as shown in Listing 45-6. By defining your own exception class, any user of rational can easily distinguish its exceptions from other exceptions.

Listing 45-6. Throwing an Exception for a Zero Denominator

```
#ifndef RATIONAL_HPP_
#define RATIONAL_HPP_

#include <stdexcept>
#include <string>

class rational
{
public:
  class zero_denominator : public std::logic_error
  {
  public:
    zero_denominator(std::string const& what_arg) : logic_error{what_arg} {}
  };

  rational() : rational{0} {}
  rational(int num) : numerator_{num}, denominator_{1} {}
  rational(int num, int den) : numerator_{num}, denominator_{den}
  {
    if (denominator_ == 0)
      throw zero_denominator{"zero denominator"};
    reduce();
  }
... omitted for brevity ...
};
#endif
```

Notice how the zero_denominator class nests within the rational class. The nested class is a perfectly ordinary class. It has no connection with the outer class (as with a Java inner class), except the name. The nested class gets no special access to private members in the outer class, nor does the outer class get special access to the nested class name. The usual rules for access levels determine the accessibility of the nested class. Some nested classes are private helper classes, so you would declare them in a private section of the outer class definition. In this case, zero_denominator must be public, so that callers can use the class in exception handlers.

To use a nested class name outside the outer class, you must use the outer class and the nested class names, separated by a scope operator (::). The nested class name has no significance outside of the outer class's scope. Thus, nested classes help avoid name collisions. They also provide clear documentation for the human reader who sees the type in an exception handler:

```
catch (rational::zero_denominator const& ex) {
  std::cerr << "zero denominator in rational number\n";
}
```

Find all other places in the **rational** class that have to check for a zero denominator and add appropriate error-checking code to throw **zero_denominator**.

All roads lead to reduce(), so one approach is to replace the assertion with a check for a zero denominator, and throw the exception there. You don't have to modify any other functions, and even the extra check in the constructor (illustrated in Listing 45-6) is unnecessary. Listing 45-7 shows the latest implementation of reduce().

Listing 45-7. Checking for a Zero Denominator in reduce()

```
void rational::reduce()
{
  if (denominator_ == 0)
    throw zero_denominator{"denominator is zero"};
  if (denominator_ < 0)
  {
    denominator_ = -denominator_;
    numerator_ = -numerator_;
  }
  int div{gcd(numerator_, denominator_)};
  numerator_ = numerator_ / div;
  denominator_ = denominator_ / div;
}
```

Don't Throw Exceptions

Certain functions should never throw an exception. For example, the `numerator()` and `denominator()` functions simply return an integer. There is no way they can throw an exception. If the compiler knows that the functions never throw an exception, it can generate more efficient object code. With these specific functions, the compiler probably expands the functions inline to access the data members directly, so in theory, it doesn't matter. But maybe you decide not to make the functions inline (for any of the reasons listed in Exploration 30). You still want to be able to tell the compiler that the functions cannot throw any exceptions. Enter the `noexcept` qualifier.

To tell the compiler that a function does not throw an exception, add the `noexcept` qualifier after the function parameters (after `const` but before `override`).

```
int numerator() const noexcept;
```

What happens if you break the contact? **Try it.** Write a program that calls a trivial function that is qualified as `noexcept`, but throws an exception. Try to catch the exception in `main()`. **What happens?**

____

If your program looks anything like mine in Listing 45-8, the `catch` is supposed to catch the exception, but it doesn't. The compiler trusted `noexcept` and did not generate the normal exception-handling code. As a result, when `function()` throws an exception, the only thing the program can do is terminate immediately.

Listing 45-8. Throwing an Exception from a `noexcept` Function

```
#include <iostream>
#include <exception>

void function() noexcept
{
  throw std::exception{};
}

int main()
{
  try {
    function();
  } catch (std::exception const& ex) {
    std::cout << "Gotcha!\n";
  }
}
```

So you must use `noexcept` judiciously. If function `a()` calls only functions that are marked `noexcept`, the author of `a()` might decide to make `a() noexcept` too. But if one of those functions, say, `b()`, changes and is no longer `noexcept`, then `a()` is in trouble. If `b()` throws an exception, the program unceremoniously terminates. So use `noexcept` only if you can guarantee that the function cannot throw an exception now and will never change in the future to throw an exception. So it is probably safe for `numerator()` and `denominator()` to be `noexcept` in the `rational` class, as well as the default and single-argument constructors, but I can't think of any other member function that can be `noexcept`.

Exceptional Advice

The basic mechanics of exceptions are easy to grasp, but their proper use is more difficult. The applications programmer has three distinct tasks: catching exceptions, throwing exceptions, and avoiding exceptions.

You should write your programs to catch all exceptions, even the unexpected ones. One approach is for your `main` program to have a master `try` statement around the entire program body. Within the program, you might use targeted `try` statements to catch specific exceptions. The closer you are to the source of the exception, the more contextual information you have, and the better you can ameliorate the problem, or at least present the user with more useful messages.

This outermost `try` statement catches any exceptions that other statements miss. It is a last-ditch attempt to present a coherent and helpful error message before the program terminates abruptly. At a minimum, tell the user that the program is terminating because of an unexpected exception.

In an event-driven program, such as a GUI application, exceptions are more problematic. The outermost `try` statement shuts down the program, closing all windows. Most event handlers should have their own `try` statement to handle exceptions for that particular menu pick, keystroke event, and so on.

Within the body of your program, better than catching exceptions is avoiding them. Use the `at` member function to access elements of a vector, but you should write the code so you are confident that the index is always valid. Index and length exceptions are signs of programmer error.

When writing low-level code, throw exceptions for most error situations that should not happen or that otherwise reflect programmer error. Some error conditions are especially dangerous. For example, in the `rational` class, a denominator should never be zero or negative after `reduce()` returns. If a condition arises when the denominator is indeed zero or negative, the internal state of the program is corrupt. If the program were to attempt a graceful shutdown, saving all files, etc., it might end up writing bad data to the files. Better to terminate immediately and rely on the most recent backup copy, which your program made while its state was still known to be good. Use assertions, not exceptions, for such emergencies.

Ideally, your code should validate user input, check vector indices, and make sure all arguments to all functions are valid before calling the functions. If anything is invalid, your program can tell the user with a clear, direct message and avoid exceptions entirely. Exceptions are a safety net when your checks fail or you forget to check for certain conditions.

As a general rule, libraries should throw exceptions, not catch them. Applications tend to catch exceptions more than throw them. As programs grow more complex, I will highlight situations that call for exceptions, throwing or catching.

Now that you know how to write classes, overload operators, and handle errors, you need only learn about some additional operators before you can start implementing fully functional classes of your own. The next Exploration revisits some familiar operators and introduces a few new ones.

EXPLORATION 46

More Operators

C++ has lots of operators. Lots and lots. So far, I've introduced the basic operators that you require for most programs: arithmetic, comparison, assignment, subscript, and function call. Now it's time to introduce some more: additional assignment operators, the conditional operator (which is like having an `if` statement in the middle of an expression), and the comma operator (most often used in `for` loops).

Conditional Operator

The conditional operator is a unique entry in the C++ operator bestiary, being a ternary operator, that is, an operator that takes three operands.

```
condition ? true-part : false-part
```

The *condition* is a Boolean expression. If it evaluates to true, the result of the entire expression is the *true-part*. If the condition is false, the result is the *false-part*. As with an `if` statement, only one part is evaluated; the branch not taken is skipped. For example, the following statement is perfectly safe:

```
std::cout << (x == 0 ? 0 : y / x);
```

If `x` is zero, the `y / x` expression is not evaluated, and division by zero never occurs. The conditional operator has very low precedence, so you often see it written inside parentheses. A conditional expression can be the source of an assignment expression. So the following expression assigned 42 or 24 to `x`, depending on whether `test` is true.

```
x = test ? 42 : 24;
```

And an assignment expression can be the *true-part* or *false-part* of a conditional expression. That is, the following expression:

```
x ? y = 1 : y = 2;
```

is parsed as

```
x ? (y = 1) : (y = 2);
```

The *true-part* and *false-part* are expressions that have the same or compatible types, that is, the compiler can automatically convert one type to the other, ensuring that the entire conditional expression has a well-defined type.

For example, you can mix an integer and a floating-point number; the expression result is a floating-point number. The following statement prints 10.000000 if x is positive:

```
std::cout << std::fixed << (x > 0 ? 10 : 42.24) << '\n';
```

Do not use the conditional operator as a replacement for if statements. If you have a choice, use an if statement, because a statement is almost always easier to read and understand than a conditional expression. Use conditional expressions in situations when if statements are infeasible. Initializing a data member in a constructor, for example, does not permit the use of an if statement. Although you can use a member function for complicated conditions, you can also use a conditional expression for simple conditions.

The rational class (last seen in Exploration 45), for example, takes a numerator and a denominator as constructor arguments. The class ensures that its denominator is always positive. If the denominator is negative, it negates the numerator and denominator. In past Explorations, I loaded the reduce() member function with additional responsibilities, such as checking for a zero denominator and a negative denominator to reverse the signs of the numerator and denominator. This design has the advantage of centralizing all code needed to convert a rational number to canonical form. An alternate design is to separate the responsibility and let the constructor check the denominator prior to calling reduce(). If the denominator is zero, the constructor throws an exception; if the denominator is negative, the constructor negates the numerator and the denominator. This alternative design makes reduce() simpler, and simple functions are less error prone than complicated functions. Listing 46-1 shows how you can do this using conditional operators.

Listing 46-1. Using Conditional Expressions in a Constructor's Initializer

```
/// Construct a rational object from a numerator and a denominator.
/// If the denominator is zero, throw zero_denominator. If the denominator
/// is negative, normalize the value by negating the numerator and denominator.
/// @post denominator_ > 0
/// @throws zero_denominator
rational::rational(int num, int den)
: numerator_{den < 0 ? -num : num},
  denominator_{den == 0 ? throw zero_denominator("0 denominator") :
                        (den < 0 ? -den : den)}
{
  reduce();
}
```

A throw expression has type void, but the compiler knows it doesn't return, so you can use it as one (or both) of the parts of a conditional expression. The type of the overall expression is that of the non-throwing part (or void, if both parts throw an exception).

In other words, if den is zero, the *true-part* of the expression throws an exception. If the condition is false, the *false-part* executes, which is another conditional expression, which evaluates the absolute value of den. The initializer for the numerator also tests den, and if negative, it negates the numerator too.

Like me, you might find that the use of conditional expressions makes the code harder to read. The conditional operator is widely used in C++ programs, so you must get used to reading it. If you decide that the conditional expressions are just too complicated, write a separate, private member function to do the work, and initialize the member by calling the function, as shown in Listing 46-2.

Listing 46-2. Using a Function and Conditional Statements Instead of Conditional Expressions

```
/// Construct a rational object from a numerator and a denominator.
/// If the denominator is zero, throw zero_denominator. If the denominator
/// is negative, normalize the value by negating the numerator and denominator.
/// @post denominator_ > 0
/// @throws zero_denominator
rational::rational(int num, int den)
: numerator_{den < 0 ? -num : num}, denominator_{init_denominator(den)}
{
  reduce();
}

/// Return an initial value for the denominator_ member. This function is used
/// only in a constructor's initializer list.
int rational::init_denominator(int den)
{
  if (den == 0)
    throw zero_denominator("0 denominator");
  else if (den < 0)
    return -den;
  else
    return den;
}
```

When writing new code, use the technique that you like best, but get used to reading both programming styles.

Short-Circuit Operators

C++ lets you overload the and and or operators, but you must resist the temptation. By overloading these operators, you defeat one of their key benefits: short-circuiting.

Recall from Exploration 12 that the and and or operators do not evaluate their right-hand operands if they don't have to. That's true of the built-in operators, but not if you overload them. When you overload the Boolean operators, they become normal functions, and C++ always evaluates function arguments before calling a function. Therefore, overloaded and and or operators behave differently from the built-in operators, and this difference makes them significantly less useful.

■ **Tip** Do not overload the and and or operators.

Comma Operator

The comma (,) serves many roles: it separates arguments in a function call, parameters in a function declaration, declarators in a declaration, and initializers in a constructor's initializer list. In all these cases, the comma is a punctuator, that is, a symbol that is part of the syntax that serves only to show where one thing (argument, declarator, etc.) ends and another thing begins. It is also an operator in its own right, which is a completely different use for the same symbol. The comma as operator separates two expressions; it causes the left-hand operand to be evaluated, and then the right-hand operand is evaluated, which becomes the result of the entire expression. The value of the left-hand operand is ignored.

At first, this operator seems a little pointless. After all, what's the purpose of writing, say,

```
x = 1 + 2, y = x + 3, z = y + 4
```

instead of

```
x = 1 + 2;
y = x + 3;
z = y + 4;
```

The comma operator is not meant to be a substitute for writing separate statements. There is one situation, however, when multiple statements are not possible, but multiple expressions have to be evaluated. I speak of none other than the `for` loop.

Suppose you want to implement the `search` algorithm. Implementing a fully generic algorithm requires techniques that you haven't learned yet, but you can write this function so that it works with the iterators of a `vector<int>`. The basic idea is simple, `search` looks through a search range, trying to find a sequence of elements that are equal to elements in a match range. It steps through the search range one element at a time, testing whether a match starts at that element. If so, it returns an iterator that refers to the start of the match. If no match is found, `search` returns the end iterator. To check for a match, use a nested loop to compare successive elements in the two ranges. Listing 46-3 shows one way to implement this function.

Listing 46-3. Searching for a Matching Sub-range in a Vector of Integers

```
#include <vector>

typedef std::vector<int>::iterator viterator;
typedef std::vector<int>::difference_type vdifference;

viterator search(viterator first1,viterator last1,viterator first2,viterator last2)
{
  // s1 is the size of the untested portion of the first range
  // s2 is the size of the second range
  // End the search when s2 > s1 because a match is impossible if the remaining
  // portion of the search range is smaller than the test range. Each iteration
  // of the outer loop shrinks the search range by one, and advances the first1
  // iterator. The inner loop searches for a match starting at first1.
  for (vdifference s1(last1-first1), s2(last2-first2); s2 <= s1; --s1, ++first1)
  {
    // Is there a match starting at first1?
    viterator f2(first2);
    for (viterator f1(first1);
         f1 != last1 and f2 != last2 and *f1 == *f2;
         ++f1, ++f2)
     {
        // The subsequence matches so far, so keep checking.
        // All the work is done in the loop header, so the body is empty.
     }
     if (f2 == last2)
       return first1;     // match starts at first1
  }
  // no match
  return last1;
}
```

The boldface lines demonstrate the comma operator. The initialization portion of the first `for` loop does not invoke the comma operator. The comma in the declaration is only a separator between declarators. The comma operator appears in the postiteration part of the loops. Because the postiteration part of a `for` loop is an expression, you cannot use multiple statements to increment multiple objects. Instead, you have to do it in a single expression. Hence, the need for the comma operator.

On the other hand, some programmers prefer to avoid the comma operator, because the resulting code can be hard to read. **Rewrite Listing 46-3 so that it does not use the comma operator. Which version of the function do you prefer?** ________________ Listing 46-4 shows my version of `search` without the comma operator.

Listing 46-4. The `search` Function Without Using the Comma Operator

```
#include <vector>

typedef std::vector<int>::iterator viterator;
typedef std::vector<int>::difference_type vdifference;

viterator search(viterator first1,viterator last1,viterator first2,viterator last2)
{
  // s1 is the size of the untested portion of the first range
  // s2 is the size of the second range
  // End the search when s2 > s1 because a match is impossible if the remaining
  // portion of the search range is smaller than the test range. Each iteration
  // of the outer loop shrinks the search range by one, and advances the first1
  // iterator. The inner loop searches for a match starting at first1.
  for (vdifference s1(last1-first1), s2(last2-first2); s2 <= s1; --s1)
  {
    // Is there a match starting at first1?
    viterator f2(first2);
    for (viterator f1(first1); f1 != last1 and f2 != last2 and *f1 == *f2; )
    {
      ++f1;
      ++f2;
    }
    if (f2 == last2)
      return first1;     // match starts at first1
    ++first1;
  }
  // no match
  return last1;
}
```

The comma operator has very low precedence, even lower than assignment and the conditional operator. If a loop has to advance objects by steps of 2, for example, you can use assignment expressions with the comma operator.

```
for (int i{0}, j{size-1}; i < j; i += 2, j -= 2) do_something(i, j);
```

By the way, C++ lets you overload the comma operator, but you shouldn't take advantage of this feature. The comma is so basic, C++ programmers quickly grasp its standard use. If the comma does not have its usual meaning, readers of your code will be confused, bewildered, and stymied when they try to understand it.

Arithmetic Assignment Operators

In addition to the usual arithmetic operators, C++ has assignment operators that combine arithmetic with assignment: +=, -=, *=, /=, and %=. The assignment operator `x += y` is shorthand for `x = x + y`, and the same applies to the other special assignment operators. Thus, the following three expressions are all equivalent if `x` has a numeric type:

```
x = x + 1;
x += 1;
++x;
```

The advantage of the special assignment operator is that `x` is evaluated only once, which can be a boon if `x` is a complicated expression. If `data` has type `std::vector<int>`, which of the following two equivalent expressions do you find easier to read and understand?

```
data.at(data.size() / 2) = data.at(data.size() / 2) + 10;
data.at(data.size() / 2) += 10;
```

Listing 46-5 shows a sample implementation of `*=` for the `rational` class.

Listing 46-5. Implementing the Multiplication Assignment Operator

```
rational& rational::operator*=(rational const& rhs)
{
  numerator_ *= rhs.numerator();
  denominator_ *= rhs.denominator();
  reduce();
  return *this;
}
```

The return type of `operator*=` is a reference, `rational&`. The return value is `*this`. Although the compiler lets you use any return type and value, the convention is for assignment operators to return a reference to the object, that is, an lvalue. Even if your code never uses the return value, many programmers use the result of an assignment, so don't use `void` as a return type.

```
rational r;
while ((r += rational{1,10}) != 2) do_something(r);
```

Often, implementing an arithmetic operator, such as +, is easiest to do by implementing the corresponding assignment operator first. Then implement the free operator in terms of the assignment operator, as shown in Listing 46-6 for the `rational` class.

Listing 46-6. Reimplementing Multiplication in Terms of an Assignment Operator

```
rational operator*(rational const& lhs, rational const& rhs)
{
  rational result{lhs};
  result *= rhs;
  return result;
}
```

Implement the /=, +=, and -= operators for class rational. You can implement these operators in many ways. I recommend putting the arithmetic logic in the assignment operators and reimplementing the /, +, and - operators to use the assignment operators, as I did with the multiplication operators. My solution appears in Listing 46-7.

Listing 46-7. Other Arithmetic Assignment Operators

```
rational& rational::operator+=(rational const& rhs)
{
  numerator_ = numerator() * rhs.denominator() + rhs.numerator() * denominator();
  denominator_ *= rhs.denominator();
  reduce();
  return *this;
}

rational& rational::operator-=(rational const& rhs)
{
  numerator_ = numerator() * rhs.denominator() - rhs.numerator() * denominator();
  denominator_ *= rhs.denominator();
  reduce();
  return *this;
}

rational& rational::operator/=(rational const& rhs)
{
  if (rhs.numerator() == 0)
    throw zero_denominator{"divide by zero"};
  numerator_ *= rhs.denominator();
  denominator_ *= rhs.numerator();
  if (denominator_ < 0)
  {
    denominator_ = -denominator_;
    numerator_ = -numerator_;
  }
  reduce();
  return *this;
}
```

Because `reduce()` no longer checks for a negative denominator, any function that might change the denominator to negative must check. Because the denominator is always positive, you know that `operator+=` and `operator-=` cannot cause the denominator to become negative. Only `operator/=` introduces that possibility, so only that function needs to check.

Increment and Decrement

Let's add increment (++) and decrement (`--`) operators to the `rational` class. Because these operators modify the object, I suggest implementing them as member functions, although C++ lets you use free functions too. **Implement the prefix increment operator for class rational**. Compare your function with mine in Listing 46-8.

Listing 46-8. The Prefix Increment Operator for `rational`

```
rational& rational::operator++()
{
  numerator_ += denominator_;
  return *this;
}
```

I am confident that you can implement the decrement operator with no additional help. Like the arithmetic assignment operators, the prefix `operator++` returns the object as a reference.

That leaves the postfix operators. Implementing the body of the operator is easy and requires only one additional line of code. However, you must take care with the return type. The postfix operators cannot simply return `*this`, because they return the original value of the object, not its new value. Thus, these operators cannot return a reference. Instead, they must return a plain `rational` rvalue.

But how do you declare the function? A class can't have two separate functions with the same name (`operator++`) and arguments. Somehow, you need a way to tell the compiler that one implementation of `operator++` is prefix and another is postfix.

The solution is that when the compiler calls a custom postfix increment or decrement operator, it passes the integer `0` as an extra argument. The postfix operators don't need the value of this extra parameter; it's just a placeholder to distinguish prefix from postfix.

Thus, when you declare `operator++` with an extra parameter of type `int`, you are declaring the postfix operator. When you declare the operator, omit the name for the extra parameter. That tells the compiler that the function doesn't use the parameter, so the compiler won't bother you with messages about unused function parameters. **Implement the postfix increment and decrement operators for** `rational`. Listing 46-9 shows my solution.

Listing 46-9. Postfix Increment and Decrement Operators

```
rational rational::operator++(int)
{
  rational result{*this};
  numerator_ += denominator_;
  return result;
}

rational rational::operator--(int)
{
  rational result{*this};
  numerator_ -= denominator_;
  return result;
}
```

Once all the dust settles from our rehabilitation project, behold the new, improved `rational` class definition in Listing 46-10.

Listing 46-10. The `rational` Class Definition

```
#ifndef RATIONAL_HPP_
#define RATIONAL_HPP_

#include <iostream>
#include <stdexcept>
#include <string>
```

```
/// Represent a rational number (fraction) as a numerator and denominator.
class rational
{
public:
  class zero_denominator : public std::logic_error
  {
  public:
    zero_denominator(std::string const& what) : logic_error{what} {}
  };
  rational(): rational{0} {}
  rational(int num): numerator_{num}, denominator_{1} {}
  rational(int num, int den);
  rational(double r);

  int numerator()              const { return numerator_; }
  int denominator()            const { return denominator_; }
  float as_float()             const;
  double as_double()           const;
  long double as_long_double() const;

  rational& operator=(int); // optimization to avoid an unneeded call to reduce()
  rational& operator+=(rational const& rhs);
  rational& operator-=(rational const& rhs);
  rational& operator*=(rational const& rhs);
  rational& operator/=(rational const& rhs);
  rational& operator++();
  rational& operator--();
  rational operator++(int);
  rational operator--(int);

private:
  /// Reduce the numerator and denominator by their GCD.
  void reduce();
  /// Reduce the numerator and denominator, and normalize the signs of both,
  /// that is, ensure denominator is not negative.
  void normalize();
  /// Return an initial value for denominator_. Throw a zero_denominator
  /// exception if @p den is zero. Always return a positive number.
  int init_denominator(int den);
  int numerator_;
  int denominator_;
};

/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
int gcd(int n, int m);

rational abs(rational const& r);
rational operator-(rational const& r);
rational operator+(rational const& lhs, rational const& rhs);
rational operator-(rational const& lhs, rational const& rhs);
rational operator*(rational const& lhs, rational const& rhs);
rational operator/(rational const& lhs, rational const& rhs);
```

```
bool operator==(rational const& a, rational const& b);
bool operator<(rational const& a, rational const& b);

inline bool operator!=(rational const& a, rational const& b)
{
  return not (a == b);
}

inline bool operator<=(rational const& a, rational const& b)
{
  return not (b < a);
}

inline bool operator>(rational const& a, rational const& b)
{
  return b < a;
}

inline bool operator>=(rational const& a, rational const& b)
{
  return not (b > a);
}

std::istream& operator>>(std::istream& in, rational& rat);
std::ostream& operator<<(std::ostream& out, rational const& rat);

#endif
```

The next Exploration is your second project. Now that you know about classes, inheritance, operator overloading, and exceptions, you are ready to tackle some serious C++ coding.

EXPLORATION 47

Project 2: Fixed-Point Numbers

Your task for Project 2 is to implement a simple fixed-point number class. The class represents fixed-point numbers using an integer type. The number of places after the decimal point is a fixed constant, four. For example, represent the number 3.1415 as the integer 31415 and 3.14 as 31400. You must overload the arithmetic, comparison, and I/O operators to maintain the fixed-point fiction.

Name the class `fixed`. It should have the following public members:

`value_type`

A `typedef` for the underlying integer type, such as `int` or `long`. By using the `value_type` typedef throughout the `fixed` class, you can easily switch between `int` and `long` by changing only the declaration of `value_type`.

places

A `static const int` equal to 4, or the number of places after the decimal point. By using a named constant instead of hard-coding the value 4, you can easily change the value to 2 or something else in the future.

places10

A `static const int` equal to 10^{places}, or the scale factor for the fixed-point values. Divide the internal integer by `places10` to obtain the true value. Multiply a number by `places10` to scale it to an integer that the `fixed` object stores internally.

`fixed()`

Default constructor.

`fixed(value_type integer, value_type fraction)`

A constructor to make a fixed-point value from an integer part and a fractional part. For example, to construct the fixed-point value 10.0020, use `fixed{10, 20}`.

Throw `std::invalid_argument` if `fraction < 0`. If `fraction >= places10`, then the constructor should discard digits to the right, rounding off the result. For example, `fixed{3, 14159} == fixed{3, 1416}` and `fixed{31, 415926} == fixed{31, 4159}`.

fixed(double val)

A constructor to make a fixed-point value from a floating-point number. Round off the fraction and discard excess digits. Thus, `fixed{12.3456789} == fixed{12, 3456789} == fixed{12, 3457}`.

Implement the arithmetic operators, arithmetic assignment operators, comparison operators, and I/O operators. Don't concern yourself with overflow. Do your best to check for errors when reading fixed-point numbers. Be sure to handle integers without decimal points (`42`) and values with too many decimal points (`3.14159`).

Implement a member function to convert the fixed-point value to `std::string`.

to_string()

Convert the value to a string representation; e.g., 3.1416 becomes `"3.1416"` and -21 becomes `"-21.0000"`.

To convert to an integer means discarding information. To make it abundantly clear to the user, call the function `round()`, to emphasize that the fixed-point value must be rounded off to become an integer.

round()

Round off to the nearest integer. If the fractional part is exactly 5000, round to the nearest even integer (banker's rounding). Be sure to handle negative and positive numbers.

Other useful member functions give you access to the raw value (good for debugging, implementing additional operations, etc.) or the parts of the fixed-point value: the integer part and the fractional part.

integer()

Return just the integer part, without the fractional part.

fraction()

Return just the fraction part, without the integer part. The fraction part is always in the range [`0`, `places10`].

Be sure to write a header file (`fixed.hpp`) with `#include` guards. Write a separate implementation file (`fixed.cpp`). Decide which member functions should be inline (if any), and be sure to define all inline functions in `fixed.hpp`, not `fixed.cpp`. After you finish, review your solution carefully and run some tests, comparing your results to mine, which you can download from the book's web site.

If you need help testing your code, try linking your `fixed.cpp` file with the test program in Listing 47-1. The test program makes use of the `test` and `test_equal` functions, declared in `test.hpp`. The details are beyond the scope of this book. Just call `test` with a Boolean argument. If the argument is true, the test passed. Otherwise, the test failed, and `test` prints a message. The `test_equal` function takes two arguments and prints a message if they are not equal. Thus, if the program produces no output, all tests passed.

Listing 47-1. Testing the `fixed` Class

```
#include <iostream>
#include <sstream>
#include <stdexcept>

#include "test.hpp"
#include "fixed.hpp"
```

```
int main()
{
  fixed f1{};
  test_equal(f1.value(), 0);
  fixed f2{1};
  test_equal(f2.value(), 10000);
  fixed f3{3, 14162};
  test_equal(f3.value(), 31416);
  fixed f4{2, 14159265};
  test_equal(f4.value(), 21416);
  test_equal(f2 + f4, f1 + f3);
  test(f2 + f4 <= f1 + f3);
  test(f2 + f4 >= f1 + f3);
  test(f1 < f2);
  test(f1 <= f2);
  test(f1 != f2);
  test(f2 > f1);
  test(f2 >= f1);
  test(f2 != f1);

  test_equal(f2 + f4, f3 - f1);
  test_equal(f2 * f3, f3);
  test_equal(f3 / f2, f3);
  f4 += f2;
  test_equal(f3, f4);
  f4 -= f1;
  test_equal(f3, f4);
  f4 *= f2;
  test_equal(f3, f4);
  f4 /= f2;
  test_equal(f3, f4);

  test_equal(-f4, f1 - f4);
  test_equal(-(-f4), f4);
  --f4;
  test_equal(f4 + 1, f3);
  f4--;
  test_equal(f4 + 2, f3);
  ++f4;
  test_equal(f4 + 1, f3);
  f4++;
  test_equal(f4, f3);
  ++f3;
  test_equal(++f4, f3);
  test_equal(f4--, f3);
  test_equal(f4++, --f3);
  test_equal(--f4, f3);

  test_equal(f4 / f3, f2);
  test_equal(f4 - f3, f1);
  test_equal(f4.to_string(), "3.1416");
  test_equal(f4.integer(), 3);
  f4 += fixed{0,4584};
  test_equal(f4, 3.6);
  test_equal(f4.integer(), 3);
  test_equal(f4.round(), 4);

  test_equal(f3.integer(), 3);
  test_equal((-f3).integer(), -3);
  test_equal(f3.fraction(), 1416);
  test_equal((-f3).fraction(), 1416);

  test_equal(fixed{7,4999}.round(), 7);
  test_equal(fixed{7,5000}.round(), 8);
  test_equal(fixed{7,5001}.round(), 8);
  test_equal(fixed{7,4999}.round(), 7);
  test_equal(fixed{8,5000}.round(), 8);
  test_equal(fixed{8,5001}.round(), 9);

  test_equal(fixed{123,2345500}, fixed(123,2346));
  test_equal(fixed{123,2345501}, fixed(123,2346));
  test_equal(fixed{123,2345499}, fixed(123,2345));
  test_equal(fixed{123,2346500}, fixed(123,2346));
  test_equal(fixed{123,2346501}, fixed(123,2347));
  test_equal(fixed{123,2346499}, fixed(123,2346));
  test_equal(fixed{123,2346400}, fixed(123,2346));
  test_equal(fixed{123,2346600}, fixed(123,2347));

  test_equal(fixed{-7,4999}.round(), -7);
  test_equal(fixed{-7,5000}.round(), -8);
  test_equal(fixed{-7,5001}.round(), -8);
  test_equal(fixed{-7,4999}.round(), -7);
  test_equal(fixed{-8,5000}.round(), -8);
  test_equal(fixed{-8,5001}.round(), -9);

  test_equal(fixed{-3.14159265}.value(), -31416);
  test_equal(fixed{123,456789}.value(), 1234568);
  test_equal(fixed{123,4}.value(), 1230004);
  test_equal(fixed{-10,1111}.value(), -101111);

  std::ostringstream out{};
  out << f3 << " 3.14159265 " << fixed(-10,12) << " 3 421.4 end";
  fixed f5{};
  std::istringstream in{out.str()};
  test(in >> f5);
  test_equal(f5, f3);
  test(in >> f5);
  test_equal(f5, f3);
  test(in >> f5);
  test_equal(f5.value(), -100012);
  test(in >> f5);
  test_equal(f5.value(), 30000);
  test(in >> f5);
  test_equal(f5.value(), 4214000);
  test(not (in >> f5));

  test_equal(fixed{31.4159265}, fixed{31, 4159});
  test_equal(fixed{31.41595}, fixed{31, 4160});

  bool okay{false};
  try {
    fixed f6{1, -1};
  } catch (std::invalid_argument const& ex) {
    okay = true;
  } catch (...) {
  }
  test(okay);
}
```

If you need a hint, I implemented `fixed` so that it stores a single integer, with an implicit decimal place `places10` positions from the right. Thus, I store the value 1 as 10000. Addition and subtraction are easy. When multiplying or dividing, you have to scale the result. (Even better is to scale the operands prior to multiplication, which avoids some overflow situations, but you have to be careful about not losing precision).

EXPLORATION 48

Function Templates

You saw in Exploration 24 that the magic of overloading lets C++ implement an improved interface to the absolute value function. Instead, of three different names (abs, labs, and fabs), C++ has a single name for all three functions. Overloading helps the programmer who needs to call the abs function, but it doesn't help the implementer much, who still has to write three separate functions that all look and act the same. Wouldn't it be nice if the library author could write the abs function once instead of three times? After all, the three implementations may be identical, differing only in the return type and parameter type. This Exploration introduces this style of programming, called generic programming.

Generic Functions

Sometimes, you want to provide overloaded functions for integer and floating-point types, but the implementation is essentially the same. Absolute value is one example; for any type T, the function looks the same (I'm using the name absval, to avoid any confusion or conflict with the standard library's abs), as shown in Listing 48-1.

Listing 48-1. Writing an Absolute Value Function

```
T absval(T x)
{
  if (x < 0)
    return -x;
  else
    return x;
}
```

Substitute int for T, double for T, or use any other numeric type. You can even substitute rational for T, and the absval function still works the way you expect it to. So why waste your precious time writing, rewriting, and re-rewriting the same function? With a simple addition to the function definition, you can turn the function into a generic function, that is, a function that works with any suitable type T, which you can see in Listing 48-2.

Listing 48-2. Writing a Function Template

```
template<class T>
T absval(T x)
{
  if (x < 0)
    return -x;
  else
    return x;
}
```

The first line is the key. The template keyword means that what follows is a template, in this case, a *function template* definition. The angle brackets delimit a comma-separated list of template parameters. A function template is a pattern for creating functions, according to the parameter type, T. Within the function template definition, T represents a type, potentially any type. The caller of the absval function determines the template argument that will substitute for T.

When you define a function template, the compiler remembers the template but does not generate any code. The compiler waits until you use the function template, and then it generates a real function. You can imagine the compiler taking the source text of the template, substituting the template argument, such as int, for the template parameter, T, and then compiling the resulting text. The next section tells you more about how to use a function template.

Using Function Templates

Using a function template is easy, at least in most situations. Just call the absval function, and the compiler will automatically determine the template argument based on the function argument type. It might take you a little while to get comfortable with the notion of template parameters and template arguments, which are quite different from function parameters and function arguments.

In the case of absval, the template parameter is T, and the template argument must be a type. You can't pass a type as a function argument, but templates are different. You aren't really "passing" anything in the program. Template magic occurs at compile time. The compiler sees the template definition of absval, and later it sees an invocation of the absval function template. The compiler examines the type of the function argument and, from the function argument's type, determines the template argument. The compiler substitutes that template argument for T and generates a new instance of the absval function, customized for the template argument type. Thus, in the following example, the compiler sees that x has type int, so it substitutes int for T.

```
int x{-42};
```

`int y{absval(x)};` The compiler generates a function just as though the library implementer had written the following:

```
int absval(int x)
{
  if (x < 0)
    return -x;
  else
    return x;
}
```

Later, in the same program, perhaps you call absval on a rational object:

```
rational r{-420, 10};
rational s{absval(r)};
```

The compiler generates a new instance of absval:

```
rational absval(rational x)
{
  if (x < 0)
    return -x;
  else
    return x;
}
```

In this new instance of absval, the < operator is the overloaded operator that takes rational arguments. The negation operator is also a custom operator that takes a rational argument. In other words, when the compiler generates an instance of absval, it does so by compiling the source code pretty much as the template author wrote it.

Write a sample program that contains the `absval` function template definition and some test code to call `absval` with a variety of argument types. Convince yourself that function templates actually work. Compare your test program with mine in Listing 48-3.

Listing 48-3. Testing the `absval` Function Template

```
#include <iostream>
#include "rational.hpp"  // Listing 46-10


template<class T>
T absval(T x)
{
  if (x < 0)
    return -x;
  else
    return x;
}

int main()
{
  std::cout << absval(-42) << '\n';
  std::cout << absval(-4.2) << '\n';
  std::cout << absval(42) << '\n';
  std::cout << absval(4.2) << '\n';
  std::cout << absval(-42L) << '\n';
  std::cout << absval(rational{42, 5}) << '\n';
  std::cout << absval(rational{-42, 5}) << '\n';
  std::cout << absval(rational{42, -5}) << '\n';
}
```

Writing Function Templates

Writing function templates is harder than writing ordinary functions. When you write a template such as absval, the problem is that you don't know what type or types T will actually be. So, the function must be generic. The compiler will prevent you from using certain types for T. Its restrictions are implicit by the way the template body uses T.

In particular, absval imposes the following restrictions on T:

- *T must be copyable.* That means you must be able to copy an object of type T, so arguments can be passed to the function and a result can be returned. If T is a class type, the class must have an accessible copy constructor, that is, the copy constructor must not be private.
- *T must be comparable with 0 using the < operator*. You might overload the < operator, or the compiler can convert 0 to T, or T to an int.
- *Unary* operator- *must be defined for an operand of type T*. The result type must be T or something the compiler can convert automatically to T.

The built-in numeric types all meet these requirements. The rational type also meets these requirements, because of the custom operators it supports. The string type, just to name one example, does not, because it lacks a comparison operator when the right-hand operand is an integer, and it lacks a unary negation (-) operator. Suppose you tried to call absval on a string.

```
std::string test{"-42"};
std::cout << absval(test) << '\n';
```

What do you think would happen?

____

____

Try it. What really happens?

____

____

The compiler complains about the lack of the comparison and negation operators for std::string. One difficulty in delivering helpful error messages when working with templates is whether to give you the line number where the template is used or the line number in the template definition. Sometimes, you will get both. Sometimes, the compiler cannot report an error in the template definition unless you try to use the template. Other errors it can report immediately. **Read Listing 48-4 carefully.**

Listing 48-4. Mystery Function Template

```
template<class T>
T add(T lhs, T rhs)
{
  return lhs(rhs);
}

int main()
{
}
```

What is the error? ____

Does your compiler report it? ____

Because the compiler does not know the type T, it cannot tell what lhs(rhs) means. We know that we want to use a numeric type for T, so lhs(rhs) is silly. After all, what does 3(4) mean?

How can you force your compiler to report the error?

____

Add a line of code to use the template. For example, add the following to main:

```
return add(0, 0);
```

Every compiler will now report the not-really-a-function-call expression in the template definition.

Template Parameters

Whenever you see T in a C++ program, most likely, you are looking at a template. Look backward in the source file until you find the template header, that is, the part of the declaration that starts with the template keyword. That's where you should find the template parameters. The use of T as a template parameter name is merely a convention, but its use is nearly universal. The use of class to declare T may seem a little strange, especially because you've seen several examples when the template argument is not, in fact, a class.

Instead of class to declare a template parameter type, some programmers use an alternate keyword, typename, which means the same thing in this one context. The advantage of typename over class is that it avoids any confusion over nonclass types. The disadvantage is that typename has more than one use in a template context, which can confuse human readers in more complicated template definitions. Learn to read both styles, but I prefer to use class when I write my own templates.

Sometimes, you will see parameter names that are more specific than T. If the template has more than one parameter, every parameter must have a unique name, so you will definitely see names other than T. For example, the copy algorithm is a function template with two parameters: the input iterator type and the output iterator type. The definition of copy, therefore, might look something like Listing 48-5.

Listing 48-5. One Way to Implement the copy Algorithm

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator start, InputIterator end, OutputIterator result)
{
  for ( ; start != end; ++start, ++result)
    *result = *start;
  return result;
}
```

Pretty simple, isn't it? (The real copy function is probably more complicated, with optimizations for certain types. Somewhere in the optimized implementation, however, is probably a function that looks just like Listing 48-5, albeit with different parameter names.)

When you use the copy algorithm, the compiler determines the value of InputIterator and OutputIterator according to the function argument types. As you saw with absval, the function's requirements on the template arguments are all implicit: InputIterator must allow the following operators: !=, prefix ++, and unary *. OutputIterator must allow prefix ++ and unary * operators. Now, perhaps, you can start to see why the requirements on iterators (Exploration 44) are written the way they are. Instead of mandating, say, that all iterators must derive from some base iterator class, the requirements are defined solely in terms of allowed operators.

Write a simple implementation of the find **algorithm.** This algorithm takes three arguments: the first two are input iterators that specify a range to search. The third argument is a value. Successive items in the range are compared with the value (using ==), and when an item matches, the function returns an iterator that refers to the matching position. If the item is not found, the end iterator is returned. Compare your solution with Listing 48-6.

Listing 48-6. Implementing the `find` Algorithm

```
template<class InputIterator, class T>
InputIterator find(InputIterator start, InputIterator end, T value)
{
  for ( ; start != end; ++start)
    if (*start == value)
      return start;
  return end;
}
```

Many of the standard algorithms are quite simple at their heart. Modern implementations are heavily optimized, and as is the nature of hand-optimized code, the results often bear little resemblance to the original code, and the optimized code can be much harder to read. Nonetheless, the simplicity remains in the architecture of the standard library, which relies extensively on templates.

Template Arguments

Templates are easiest to use when the compiler automatically deduces the template arguments from the function arguments. It can't always do so, however, so you might have to tell the compiler explicitly what you want. The simple form of the min, max, and mixmax standard algorithms, for example, take a single template parameter. Listing 48-7 shows one possible implementation of the min function, for reference.

Listing 48-7. The `std::min` Algorithm

```
template<class T>
T min(T a, T b)
{
  if (a < b)
    return a;
  else if (b < a)
    return b;
  else
    return a; // a and b are equivalent, so return a
}
```

If both argument types are the same, the compiler can deduce the desired type, and everything just works.

```
int x{10}, y{20}, z{std::min(x, y)};
```

On the other hand, if the function argument types are different, the compiler can't tell which type to use for the template argument.

```
int x{10};
long y{20};
std::cout << std::min(x, y); // error
```

Why is that? Suppose you wrote your own function as a non-template.

```
long my_min(long a, long b)
{
  if (a <= b)
    return a;
  else
    return b;
}
```

The compiler could handle my_min(x, y) by converting x from an int to a long. As a template, however, the compiler does not perform any automatic type conversion. The compiler cannot read your mind and know that you want the template parameter to have the type of the first function argument or the second, or sometimes the first and sometimes the second. Instead, the compiler requires you to write exactly what you mean. In this case, you can tell the compiler what type to use for the template parameter by enclosing the desired type in angle brackets.

```
int x{10};
long y{20};
std::cout << std::min<long>(x, y); // okay: compiler converts x to type long
```

If a template takes multiple arguments, separate the arguments with a comma. For example, Listing 48-8 shows the input_sum function, which reads items from the standard input and accumulates them by means of the += operator. The accumulator can have a different type than the item type. Because the item and accumulator types are not used in the function arguments, the compiler cannot deduce the parameter arguments, so you must supply them explicitly.

Listing 48-8. Multiple Template Arguments

```
#include <iostream>

template<class T, class U>
U input_sum(std::istream& in)
{
  T x{};
  U sum{0};
  while (in >> x)
    sum += x;
  return sum;
}

int main()
{
  long sum{input_sum<int, long>(std::cin)};
  std::cout << sum << '\n';
}
```

Rewrite the gcd function (from Listing 35-4) to be a function template, so that you can use the same function template for int, short, or long arguments. Compare your solution with mine in Listing 48-9. Remember to use the template parameter to declare the temporary variable.

Listing 48-9. The gcd Function Template

```
#ifndef GCD_HPP_
#define GCD_HPP_

#include <cstdlib> // for std::abs

template<class T>
/// Compute the greatest common divisor of two integers, using Euclid's algorithm.
T gcd(T n, T m)
{
  n = std::abs(n);
  while (m != 0) {
    T tmp{n % m};
    n = m;
    m = tmp;
  }
  return n;
}
#endif
```

Declarations and Definitions

I can't seem to stop talking about declarations and definitions. Templates introduce yet another twist to this plot. When you work with templates, the rules change. The compiler has to see more than just a declaration before you can use a function template. The compiler usually requires the full function template definition. In other words, if you define a template in a header file, that header file must include the body of that function template. Suppose you want to share the gcd function among many projects. Ordinarily, you would put the function declaration in a header file, say, gcd.hpp, and put the full definition in a separate source file, say, gcd.cpp.

When you convert gcd to a function template, however, you usually put the definition in the header file, as you saw in Listing 48-9. In order for the compiler to create concrete functions from the template, say, for gcd<int> or gcd<long>, it needs the body of the function template.

Member Function Templates

The rational class has three nearly identical functions: to_long_double, to_double, and to_float. They all do the same thing: divide the numerator by the denominator after converting to the destination type. Whenever you have multiple functions doing the same thing, in the same way, using the same code, you have a candidate for a template, such as the following:

```
template<class T, class R>
T convert(R const& r)
{
  return static_cast<T>(r.numerator()) / r.denominator();
}
```

As with any function template, the only requirement on R is that objects of type R have member functions named numerator( ) and denominator( ) and that these functions have return types suitable for use with operator/ (which could be overloaded). To use the convert function, you must supply the target type, T, as an explicit template argument, but you can let the compiler deduce R from the function argument:

```
rational r{42, 10};
double d{ convert<double>(r) };
```

You can omit template arguments that the compiler can deduce, starting from the rightmost argument. As you saw earlier in this Exploration, if the compiler can deduce all the arguments, you can leave out the angle brackets entirely.

A function template can be a member function too. Instead of passing the rational object as an argument, you might prefer to use a member function template, as follows:

```
rational r{42, 10};
double d{ r.convert<double>() };
```

A member function template avoids collisions with other free functions that might be named convert. For normal (non-template) functions, ordinary overloading rules often help to keep collisions to a minimum. It is unlikely that another function named convert would take a rational object as an argument. But templates are more problematic. Nothing about the template restricts the argument to rational, so the template convert collides with every other function named convert that takes one argument, even if that argument has nothing to do with rational.

The details are more complicated than I've described, but the short version will do for now as a justification for writing a member function template instead of a free function template. Listing 48-10 shows how you might write the convert member function template.

Listing 48-10. Implementing a Member Function Template

```
class rational
{
public:
  rational() : rational{0} {}
  rational(int num) : numerator_{num}, denominator_{1} {}
  rational(int num, int den);
  rational(double r);

  template<class T>
  T convert()
  const
  {
    return static_cast<T>(numerator()) / static_cast<T>(denominator());
  }

... omitted for brevity ...
};
```

Generic programming is a powerful technique, and as you learn more about it in the next several Explorations, you will see how expressive and useful this programming paradigm is.

EXPLORATION 49

Class Templates

A class can be a template , which makes all of its members templates. Every program in this book has used class templates, because much of the standard library relies on templates: the standard I/O streams, strings, vectors, and maps are all class templates. This Exploration takes a look at simple class templates.

Parameterizing a Type

Consider a simple point class, which stores an x and y coordinate. A graphics device driver might use int for the member types.

```
class point {
public:
   point(int x, int y) : x_{x}, y_{y} {}
   int x() const { return x_; }
   int y() const { return y_; }
private:
   int x_, y_;
};
```

On the other hand, a calculus tool probably prefers to use double.

```
class point {
public:
   point(double x, double y) : x_{x}, y_{y} {}
   double x() const { return x_; }
   double y() const { return y_; }
private:
   double x_, y_;
};
```

Imagine adding much more functionality to the point classes: computing distances between two point objects, rotating one point around another by some angle, etc. The more functionality you dream up, the more you must duplicate in both classes.

Wouldn't your job be simpler if you could write the point class once and use that single definition for both of these situations and for others not yet dreamed of? Templates to the rescue. Listing 49-1 shows the point class template.

Listing 49-1. The `point` Class Template

```
template<class T>
class point {
public:
   point(T x, T y) : x_{x}, y_{y} {}
   T x() const { return x_; }
   T y() const { return y_; }
   void move_to(T x, T y);           ///< Move to absolute coordinates (x, y)
   void move_by(T x, T y);           ///< Add (x, y) to current position
private:
   T x_, y_;
};

template<class T>
void point<T>::move_to(T x, T y)
{
  x_ = x;
  y_ = y;
}
```

Just as with function templates, the template keyword introduces a class template. The class template is a pattern for making classes, which you do by supplying template arguments; e.g., point<int>.

The member functions of a class template are themselves function templates, using the same template parameters, except that you supply the template arguments to the class, not the function, as you can see in the point<T>::move_to function. **Write the move_by member function.** Compare your solution with Listing 49-2.

Listing 49-2. The `move_by` Member Function

```
template<class T>
void point<T>::move_by(T x, T y)
{
  x_ += x;
  y_ += y;
}
```

Every time you use a different template argument, the compiler generates a new class instance, with new member functions. That is, point<int>::move_by is one function, and point<double>::move_by is another, which is exactly what would happen if you had written the functions by hand. If two different source files both use point<int>, the compiler and linker ensure that they share the same template instance.

Parameterizing the rational Class

A simple point class is easy. What about something more complicated, such as the rational class? Suppose someone likes your rational class but wants more precision. You decide to change the type of the numerator and denominator from int to long. Someone else then complains that rational takes up too much memory and asks for a version that uses short as the base type. You could make three copies of the source code, one each for types short, int, and long. Or you could define a class template, as illustrated by the simplified rational class template in Listing 49-3.

Listing 49-3. The rational Class Template

```
#ifndef RATIONAL_HPP_
#define RATIONAL_HPP_
template<class T>
class rational
{
public:
  typedef T value_type;
  rational() : rational{0} {}
  rational(T num) : numerator_{num}, denominator_{1} {}
  rational(T num, T den);

  void assign(T num, T den);

  template<class U>
  U convert()
  const
  {
    return static_cast<U>(numerator()) / static_cast<U>(denominator());
  }

  T numerator() const { return numerator_; }
  T denominator() const { return denominator_; }
private:
  void reduce();
  T numerator_;
  T denominator_;
};

template<class T>
rational<T>::rational(T num, T den)
: numerator_{num}, denominator_{den}
{
  reduce();
}

template<class T>
void rational<T>::assign(T num, T den)
{
  numerator_ = num;
  denominator_ = den;
  reduce();
}

template<class T>
bool operator==(rational<T> const& a, rational<T> const& b)
{
  return a.numerator() == b.numerator() and
         a.denominator() == b.denominator();
}

```

```
51 template<class T>
52 inline bool operator!=(rational<T> const& a, rational<T> const& b)
53 {
54   return not (a == b);
55 }
56
57 #endif
```

The typedef of value_type (line 7) is a useful convention. Many class templates that use a template parameter as some kind of subordinate type expose the parameter under a well-defined name. For example, vector<char>::value_type is a typedef for its template parameter, namely, char.

Look at the definition of the constructor on line 29. When you define a member outside of its class template, you have to repeat the template header. The full name of the type includes the template parameter, rational<T> in this case. Inside the class scope, use only the class name, without the template parameter. Also, once the compiler sees the fully qualified class name, it knows it is inside the class scope, and you can also use the template parameter by itself, which you can see in the parameter declarations.

Because the name T is already in use, the convert member function (line 14) needs a new name for its template parameter. U is a common convention, provided you don’t take it too far. More than two or three single-letter parameters, and you start to need more meaningful names, just to help keep straight which parameter goes with which template.

In addition to the class template itself, you have to convert all the free functions that support the rational type to be function templates. Listing 49-3 keeps things simple by showing only operator== and operator!=. Other operators work similarly.

Using Class Templates

Unlike function templates, the compiler cannot deduce the template argument of a class template. This means you must supply the argument explicitly, inside angle brackets.

```
rational<short> zero{};
rational<int> pi1{355, 113};
rational<long> pi2{80143857L, 25510582L};
```

Notice anything familiar? Does rational<int> look like vector<int>? All the collection types, such as vector and map, are class templates. The standard library makes heavy use of templates throughout, and you will discover other templates when the time is right.

If a class template takes multiple arguments, separate the arguments with a comma, as in map<long, int>. A template argument can even be another template, such as

```
std::vector<std::vector<int>> matrix;
```

Starting with rational.hpp from Listing 49-3, **add the I/O operators**. (See Listing 35-4 for the non-template versions of the operators.) Write a simple test program that reads rational objects and echoes the values, one per line, to the standard output. Try changing the template argument to different types (short, int, long). Your test program might look something like Listing 49-4.

Listing 49-4. Simple I/O Test of the `rational` Class Template

```
#include <iostream>
#include "rational.hpp"

int main()
{
  rational<int> r{};
  while (std::cin >> r)
    std::cout << r << '\n';
}
```

Now modify the test program to print only nonzero values. The program should look something like Listing 49-5.

Listing 49-5. Testing `rational` Comparison Operator

```
#include <iostream>
#include "rational.hpp"

int main()
{
  rational<int> r{};
  while (std::cin >> r)
    if (r != 0)
      std::cout << r << '\n';
}
```

Remember that with the old rational class, the compiler knew how to construct a rational object from an integer. Thus, it could convert the 0 to rational(0) and then call the overloaded == operator to compare two rational objects. So everything is fine. Right? So why doesn't it work?

Overloaded Operators

Remember from the previous Exploration that the compiler does not perform automatic type conversion for a function template. That means the compiler will not convert an int to a rational<int>. To solve this problem, you have to add some additional comparison operators, such as

```
template<class T> bool operator==(rational<T> const& lhs, T rhs);
template<class T> bool operator==(T lhs, rational<T> const& rhs);
template<class T> bool operator!=(rational<T> const&  lhs, T rhs);
template<class T> bool operator!=(T lhs, rational<T> const& rhs);
```

and so on, for all of the comparison and arithmetic operators. On the other hand, you have to consider whether that's what you really want. To better understand the limitations of this approach, go ahead and try it. You don't need all the comparison operators yet, just operator !=, so you can compile the test program. After adding the two new overloaded operator != functions, compile Listing 49-5 again, to be sure it works. Next, compile the test program with a template parameter of long. What happens?

____

____

Once again, the compiler complains that it can't find any suitable function for the != operator. The problem is that an overloaded != operator exists for the template parameter, namely, type long, but the type of the literal 0 is int, not long. You can try to solve this problem by defining operators for all the built-in types, but that quickly gets out of hand. So your choices are the following:

- Define only operators that take two rational arguments. Force the caller to convert arguments to the desired rational type.
- Define operators in triples: one that takes two rational arguments and two others that mix one rational and one base type (T).
- Define operators to cover all the bases—for the built-in types (signed char, char, short, int, long), plus some types that I haven't covered yet. Thus, each operator requires 11 functions.

You might be interested in knowing how the C++ standard library addresses this issue. Among the types in the standard library is a class template, complex, which represents a complex number. The standardization committee opted for the second choice, that is, three overloaded function templates.

```
template<class T> bool operator==(complex<T> const& a, complex<T> const& b);
template<class T> bool operator==(complex<T> const& a, T const& b);
template<class T> bool operator==(T const& a, complex<T> const& b);
```

This solution works well enough, and later in the book, you'll learn techniques to reduce the amount of work involved in defining all these functions.

Another dimension to this problem is the literal 0. Using a literal of type int is fine when you know the base type of rational is also int. How do you express a generic zero for use in a template? The same issue arises when testing for a zero denominator. That was easy when you knew that the type of the denominator was int. When working with templates, you don't know the type. Recall from Listing 45-6 that the division assignment operator checked for a zero divisor and threw an exception in that case. If you don't know the type T, how do you know how to express the value zero? You can try using the literal 0 and hope that T has a suitable constructor (single argument of type int). A better solution is to invoke the default constructor for type T, as shown in Listing 49-6.

Listing 49-6. Invoking a Default Constructor of a Template Parameter

```
template<class T>
rational<T>& rational<T>::operator/=(rational const& rhs)
{
  if (rhs.numerator() == T{})
    throw zero_denominator("divide by zero");
  numerator_ *= rhs.denominator();
  denominator_ *= rhs.numerator();
  if (denominator_ < T{})
  {
    denominator_ = -denominator_;
    numerator_ = -numerator_;
  }
  reduce();
  return *this;
}
```

If the type T is a class type, T{} yields an object that is initialized using T's default constructor. If T is a built-in type, the value of T{} is zero (i.e., 0, 0.0, or false). Initializing the local variables in the input operator is a little trickier.

Mixing Types

As you know, you can assign an int value to a long object or *vice versa*. It seems reasonable, therefore, that you should be able to assign a rational<int> value to a rational<long> object. Try it. **Write a simple program to perform an assignment that mixes base types.** Your program might look a little bit like Listing 49-7, but many other programs are equally reasonable.

Listing 49-7. Trying to Mix `rational` Base Types

```
#include "rational.hpp"

int main()
{
  rational<int> little{};
  rational<long> big{};
  big = little;
}
```

What happens when you compile your program?

____

____

The only assignment operator for the new rational class template is the compiler's implicit operator. Its parameter type is rational<T> const, so the base type of the source expression must be the same as the base type of the assignment target. You can fix this easily with a member function template. Add the following declaration to the class template:

```
template<class U>
rational& operator=(rational<U> const& rhs);
```

Inside the rational class template, the unadorned name, rational, means the same thing as rational<T>. The complete name of the class includes the template argument, so the proper name of the constructor is rational<T>. Because rational means the same as rational<T>, I was able to shorten the constructor name and many other uses of the type name throughout the class template definition. But the assignment operator's parameter is rational<U>. It uses a completely different template argument. Using this assignment operator, you can freely mix different rational types in an assignment statement.

Write the definition of the assignment operator. Don't worry about overflow that might result from assigning large values to small. It's a difficult problem and distracts from the main task at hand, which is practicing writing class templates and function templates. Compare your solution with Listing 49-8.

Listing 49-8. Defining the Assignment Operator Function Template

```
template<class T>
template<class U>
rational<T>& rational<T>::operator=(rational<U> const& rhs)
{
  assign(rhs.numerator(), rhs.denominator());
  return *this;
}
```

The first template header tells the compiler about the rational class template. The next template header tells the compiler about the assignment operator function template. Note that the compiler will be able to deduce the template argument for U from the type of the assignment source (rhs). After adding this operator to the rational class template, you should now be able to make your test program work.

Add a member template constructor that works similarly to the assignment operator. In other words, add to rational a constructor that looks like a copy constructor but isn't really. A copy constructor copies only objects of the same type, or rational<T>. This new constructor copies rational objects with a different base type, rational<U>. Compare your solution with Listing 49-9.

Listing 49-9. Defining a Member Constructor Template

```
template<class T>
template<class U>
rational<T>::rational(rational<U> const& copy)
: numerator_{copy.numerator()}, denominator_{copy.denominator()}
{}
```

Notice how the template headers stack up. The class template header comes first, followed by the constructor template header. **Finish the rational.hpp header by completing all the operators.** The new file is too big to include here, but as always, you can download the completed file from the book's web site.

Programming with templates and type parameters opens a new world of programming power and flexibility. A template lets you write a function or class once and lets the compiler generate actual functions and classes for different template arguments. Sometimes, however, one size does not fit all, and you have to grant exceptions to the rule. The next Exploration takes a look at how you do that by writing template specializations.

EXPLORATION 50

Template Specialization

The ability to write a template and then use that template multiple times, with different template arguments each time, is one of the great features of C++. Even better is the ability to carve out exceptions to the rule. That is, you can tell the compiler to use a template for most template arguments, except that for certain argument types, it should use a different template definition. This Exploration introduces this feature.

Instantiation and Specialization

Template terminology is tricky. When you use a template, it is known as *instantiating* the template. A *template instance* is a concrete function or class that the compiler creates by applying the template arguments to the template definition. Another name for a template instance is a *specialization*. Thus, rational<int> is a specialization of the template rational<>.

Therefore, specialization is the realization of a template for a specific set of template arguments. C++ lets you define a custom specialization for one particular set of template arguments; that is, you can create an exception to the rule set down by the template. When you define the specialization instead of letting the compiler instantiate the template for you, it is known as *explicit specialization*. Thus, a specialization that the compiler creates automatically would be an *implicit specialization*. (Explicit specialization is also called *full specialization*, for reasons that will become clear in the next Exploration.) For example, suppose you define a simple class template, point, to represent an (*x*, *y*) coordinate and made it into a template to accept any numeric type as the type for *x* and *y*. Because some types are large, you decide to pass values by const reference whenever possible, as shown in Listing 50-1.

Listing 50-1. The `point` Class Template

```
template<class T>
class point
{
public:
  typedef T value_type;
  point(T const& x, T const& y) : x_{x}, y_{y} {}
  point() : point{T{}, T{}} {}
  T const& x() const { return x_; }
  T const& y() const { return y_; }
  void move_absolute(T const& x, T const& y) {
    x_ = x;
    y_ = y;
  }
  void move_relative(T const& dx, T const& dy) {
    x_ += dx;
    y_ += dy;
  }
private:
  T x_;
  T y_;
};
```

The point template works with int, double, rational, and any type that behaves the same way as the built-in numeric types. If you had, say, an arbitrary-precision numeric type, you could use that too, and because such objects are potentially very large, passing by reference is a good choice for default behavior.

On the other hand, point<int> is a fairly common usage, especially in graphical user interfaces. In a mathematical context, point<double> might be more common. In either case, you might decide that passing values by reference is actually wasteful. You can define an explicit specialization for point<int> to pass arguments by value, as shown in Listing 50-2.

Listing 50-2. The `point<int>` Specialization

```
template<>
class point<int>
{
public:
  typedef int value_type;
  point(int x, int y) : x_{x}, y_{y} {}
  point() : point{0, 0} {}
  int x() const { return x_; }
  int y() const { return y_; }
  void move_absolute(int x, int y) {
    x_ = x;
    y_ = y;
  }
  void move_relative(int dx, int dy) {
    x_ += dx;
    y_ += dy;
  }
private:
  int x_;
  int y_;
};
```

Start an explicit specialization with template<> (notice the empty angle brackets), which tells the compiler that you are writing an explicit specialization. Next is the definition. Notice how the class name is the specialized template name: point<int>. That's how the compiler knows what you are specializing. Before you can specialize a template, you must tell the compiler about the class template, called the *primary* template, with a declaration of the class template name or a full definition of the class template. Typically, you would put the class template declaration, followed by its specializations, in a single header, in the right order.

Your explicit specialization completely replaces the template declaration for that template argument (or arguments; if the template takes multiple arguments, you must supply a specific value for each one). Although convention dictates that point<int> should define all the same members as the primary template, point<>, the compiler imposes no such limitation.

Write an explicit specialization for point<double>. Add a debugging statement to the primary template and to the specialization so you can prove to yourself that the compiler really does choose the specialization. Write a main program to use point<double> and point<short>, and check that the correct debugging statements execute. Listing 50-3 shows how the program looks when I write it.

Listing 50-3. Specializing `point<double>` and Testing the Code

```
#include <iostream>

template<class T>
class point
{
public:
  typedef T value_type;
  point(T const& x, T const& y) : x_{x}, y_{y} {}
  point() : point{T{}, T{}} { std::cout << "point<>()\n"; }
  T const& x() const { return x_; }
  T const& y() const { return y_; }
  void move_absolute(T const& x, T const& y) {
    x_ = x;
    y_ = y;
  }
  void move_relative(T const& dx, T const& dy) {
    x_ += dx;
    y_ += dy;
  }
private:
  T x_;
  T y_;
};

template<>
class point<double>
{
public:
  typedef double value_type;
  point(double  x, double y) : x_(x), y_(y) {}
  point() : point{0.0, 0.0} { std::cout << "point<double> specialization\n"; }
  double x() const { return x_; }
  double y() const { return y_; }
  void move_absolute(double x, double y) {
    x_ = x;
    y_ = y;
  }
  void move_relative(double dx, double dy) {
    x_ += dx;
    y_ += dy;
  }
private:
  double x_;
  double y_;
};

int main()
```

```
{
  point<short> s;
  point<double> d;
  s.move_absolute(10, 20);
  d.move_absolute(1.23, 4.56);
}
```

If you really want to get fancy, include the <typeinfo> header, and, in the primary template, call typeid(T).name() to obtain a string that describes the type T. The exact contents of the string depend on the implementation. The typeid keyword returns a typeinfo object (defined in <typeinfo>) that describes a type, or you can apply the keyword to an expression to obtain information on the expression's type. You can't do much with a typeinfo object. It's not a reflection mechanism, but you can call the name() member function to get an implementation-defined name. This is a handy debugging technique when you have a complicated template situation, and you aren't quite sure what the compiler thinks the template argument is. Thus, write the constructor as follows:

```
point() : x_(), y_() { std::cout << "point<" << typeid(T).name() << ">()\n"; }
```

One compiler I have prints

```
point<short>()
point<double> specialization
```

Another prints

```
point<s>()
point<double> specialization
```

Custom Comparators

The map container lets you provide a custom comparator. The default behavior is for map to use a template class, std::less<>, which is a functor that uses the < operator to compare keys. If you want to store a type that cannot be compared with <, you can specialize std::less for your type. For example, suppose you have a person class, which stores a person's name, address, and telephone number. You want to store a person in a map, ordered by name. All you need to do is write a template specialization, std::less<person>, as shown in Listing 50-4.

Listing 50-4. Specializing `std::less` to Compare `person` Objects by Name

```
#include <functional>
#include <iostream>
#include <map>
#include <string>

class person {
public:
   person() : name_{}, address_{}, phone_{} {}
   person(std::string const& name,
          std::string const& address,
          std::string const& phone)
   : name_{name}, address_{address}, phone_{phone}
   {}
   std::string const& name()    const { return name_; }
   std::string const& address() const { return address_; }
   std::string const& phone()   const { return phone_; }
private:
   std::string name_, address_, phone_;
};

namespace std {
   template<>
   struct less<person> {
      bool operator()(person const& a, person const& b) const {
         return a.name() < b.name();
      }
   };
}

int main()
{
   std::map<person, int> people;
   people[person{"Ray", "123 Erewhon", "555-5555"}] = 42;
   people[person{"Arthur", "456 Utopia", "123-4567"}]= 10;
   std::cout << people.begin()->first.name() << '\n';
}
```

You are allowed to specialize templates that are defined in the std namespace, but you cannot add new declarations to std. The std::less template is declared in the <functional> header. This header defines comparator templates for all the relational and equality operators, and some more besides. Consult a language reference for details. What matters right now is what the std::less primary template looks like, that is, the primary template that C++ uses when it cannot find an explicit specialization (such as std::less<person>). **Write the definition of a class template, less, that would serve as a primary template to compare any comparable objects with the < operator.** Compare your solution with Listing 50-5.

Listing 50-5. The Primary `std::less` Class Template

```
template<class T>
struct less
{
   bool operator()(T const& a, T const& b) const { return a < b; }
};
```

Take a peek into your standard library's <functional> header. It might be in another file that <functional> includes, and it might be more complicated than Listing 50-5, but you should be able to find something that you can recognize and understand. (For example, the C++ standard also mandates member typedefs for the argument and return type.)

Specializing Function Templates

You can specialize a function template, but you should prefer overloading to templates. For example, let's keep working with the template form of absval (Exploration 48). Suppose you have an arbitrary-precision integer class, integer, and it has an efficient absolute value function (that is, it simply clears the sign bit, so there's no need to compare the value with zero). Instead of the template form of absval, you want to use the efficient method for taking the absolute value of integer. Although C++ permits you to specialize the absval<> function template, a better solution is to override the absval function (not template).

```
integer absval(integer i)
{
   i.clear_sign_bit();
   return i;
}
```

When the compiler sees a call to absval, it examines the type of the argument. If the type matches the parameter type used in a non-template function, the compiler arranges to call that function. If it can't match the argument type with the parameter type, it checks template functions. The precise rules are complicated, and I will discuss them later in the book. For now, just remember that the compiler prefers non-template functions to template functions, but it will use a template function instead of a non-template function if it can't find a good match between the argument types and the parameter types of the non-template function.

Sometimes, however, you have to write a template function, even if you just want to overload the absval function. For example, suppose you want to improve the absolute value function for the rational<T> class template. There is no need to compare the entire value with zero; just compare the numerator, and avoid unnecessary multiplications.

```
template<class T>
rational<T> absval(rational<T> const& r)
{
  if (r.numerator() < 0) // to avoid unnecessary multiplications in operator<
    return -r;
  else
    return r;
}
```

When you call absval, pass it an argument in the usual way. If you pass an int, double, or other built-in numeric type, the compiler instantiates the original function template. If you pass an integer object, the compiler calls the overloaded non-template function, and if you pass a rational object, the compiler instantiates the overloaded function template.

Traits

Another use of specialization is to define a template that captures the characteristics, or *traits*, of a type. You've already seen one example of a traits template: std::numeric_limits. The <limits> header defines a class template named std::numeric_limits. The primary template is rather dull, saying that the type has zero digits of precision, a radix of zero, and so on. The only way this template makes any sense is to specialize it. Thus, the <limits> header also defines explicit specializations of the template for all the built-in types. Thus, you can discover the smallest int by calling std::numeric_limits<int>::min() or determine the floating-point radix of double with std::numeric_limits<double>::radix, and so on. Every specialization declares the same members, but with values that are particular to the specialization. (Note that the compiler does not enforce the fact that every specialization declares the same members. The C++ standard mandates this requirement for numeric_limits, and it is up to the library author to implement the standard correctly, but the compiler provides no help.)

You can define your own specialization when you create a numeric type, such as rational. Defining a template of a template involves some difficulties that I will cover in the next Exploration, so for now, go back to Listing 46-10 and the old-fashioned non-template rational class, which hard-coded int as the base type. Listing 50-6 shows how to specialize numeric_limits for this rational class.

Listing 50-6. Specializing `numeric_limits` for the `rational` Class

```
namespace std {
template<>
class numeric_limits<rational>
{
public:
  static constexpr bool is_specialized{true};
  static constexpr rational min() noexcept { return rational(numeric_limits<int>::min()); }
  static constexpr rational max() noexcept { return rational(numeric_limits<int>::max()); }
  static rational lowest() noexcept { return -max(); }
  static constexpr int digits{ 2 * numeric_limits<int>::digits };
  static constexpr int digits10{ numeric_limits<int>::digits10 };
  static constexpr int max_digits10{ numeric_limits<int>::max_digits10 };
  static constexpr bool is_signed{ true };
  static constexpr bool is_integer{ false };
  static constexpr bool is_exact{ true };
  static constexpr int radix{ 2 };
  static constexpr bool is_bounded{ true };
  static constexpr bool is_modulo{ false };
  static constexpr bool traps{ std::numeric_limits<int>::traps };

  static rational epsilon() noexcept
     { return rational(1, numeric_limits<int>::max()-1); }
  static rational round_error() noexcept
     { return rational(1, numeric_limits<int>::max()); }

  // The following are meaningful only for floating-point types.
  static constexpr int min_exponent{ 0 };
  static constexpr int min_exponent10{ 0 };
  static constexpr int max_exponent{ 0 };
  static constexpr int max_exponent10{ 0 };
  static constexpr bool has_infinity{ false };
  static constexpr bool has_quiet_NaN{ false };
  static constexpr bool has_signaling_NaN{ false };
  static constexpr float_denorm_style has_denorm {denorm_absent};
  static constexpr bool has_denorm_loss {false};
  // The following are meant only for floating-point types, but you have
  // to define them, anyway, even for nonfloating-point types. The values
  // they return do not have to be meaningful.
  static constexpr rational infinity() noexcept { return max(); }
  static constexpr rational quiet_NaN() noexcept { return rational(); }
  static constexpr rational signaling_NaN() noexcept { return rational(); }
  static constexpr rational denorm_min() noexcept { return rational(); }
  static constexpr bool is_iec559{ false };
  static constexpr bool tinyness_before{ false };
  static constexpr float_round_style round_style{ round_toward_zero };
};

} // namespace std
```

This example has a few new things. They aren't important right now, but in C++, you have to get all the tiny details right, or the compiler voices its stern disapproval. The first line, which starts namespace std, is how you put names in the standard library. You are not allowed to add new names to the standard library (although the standard does not require a compiler to issue an error message if you break this rule), but you are allowed to specialize templates that the standard library has already defined. Notice the opening curly brace for the namespace, which has a corresponding closing curly brace on the last line of the listing. (This topic will be covered in more depth in Exploration 52.)

The member functions all have noexcept between their names and bodies. This tells the compiler that the function does not throw any exceptions (recall from Exploration 45). If the function does actually throw an exception at runtime, the program terminates unceremoniously, without calling any destructors. (This topic is covered in more depth in Exploration 60.)

The constexpr specifier is similar to const, but it tells the compiler that the value is a compile-time constant. In order for a function to be constexpr, the compiler imposes a number of restrictions. In particular, the function body must be a single return statement and no other statements. Any functions that it calls must also be constexpr. Function parameter and return type must be built-in or types that can be constructed with constexpr constructors. A constexpr function cannot call a non-constexpr function. If any restriction is violated, the function cannot be declared constexpr. Thus, the gcd() function cannot be constexpr, so reduce( ) cannot be constexpr, so the two-argument constructor cannot be constexpr. The value of being able to write a function that is called at compile time is extremely useful, and we will return to constexpr in the future.

■ **Tip** When writing a template for the first time, start with a non-template version. It is much easier to debug a non-template function or class. Once you get the non-template version working, then change it into a template.

Template specialization has many other uses, but before we get carried away, the next Exploration takes a look at a particular kind of specialization, where your specialization still requires template parameters, called partial specialization.

EXPLORATION 51

Partial Template Specialization

Explicit specialization requires you to specify a template argument for every template parameter, leaving no template parameters in the template header. Sometimes, however, you want to specify only some of the template arguments, leaving one or more template parameters in the header. C++ lets you do just that and more, but only for class templates, as this Exploration describes.

Degenerate Pairs

The standard library defines the std::pair<T, U> class template in the <utility> header. This class template is a trivial holder of a pair of objects. The template arguments specify the types of these two objects. Listing 51-1 depicts a common definition of this simple template. (I omitted some members that involve more advanced programming techniques, just to keep this Exploration manageable.)

Listing 51-1. The `std::pair` Class Template

```
template<class T, class U>
struct pair
{
   typedef T first_type;
   typedef U second_type;
   T first;
   U second;
   pair();
   pair(T const& first, U const& second);
   template<class T2, class U2>
   pair(pair<T2, U2> const& other);
};
```

As you can tell, the pair class template doesn't do much. The std::map class template can use std::pair to store keys and values. A few functions, such as std::equal_range, return a pair in order to return two pieces of information. In other words, pair is a useful, if dull, part of the standard library.

What happens if T or U is void?

____

Although void has popped up here and there, usually as a function's return type, I haven't discussed it much. The void type means "no type." That's useful for returning from a function, but you cannot declare an object with void type, nor does the compiler permit you to use void for a data member. Thus, pair<int, void> results in an error.

As you start to use templates more and more, you will find yourself in unpredictable situations. A template contains a template, which contains another template, and suddenly you find a template, such as pair, being instantiated with template arguments that you never imagined before, such as void. So let's add specializations for pair that permit one or two void template arguments, just for the sake of completeness. (The standard permits specializations of library templates only if a template argument is a user-defined type. Therefore, specializing pair for the void type results in undefined behavior. If your compiler is picky, you can copy the definition of pair out of the <utility> header and into your own file, using a different namespace. Then proceed with the experiment. Most readers will be able to work in the std namespace without incurring the wrath of the compiler, at least to complete this exercise.)

Write an explicit specialization for pair<void, void>. It cannot store anything, but you can declare objects of type pair<void, void>. Compare your solution with Listing 51-2.

Listing 51-2. Specializing `pair<>` for Two `void` Arguments

```
template<>
struct pair<void, void>
{
   typedef void first_type;
   typedef void second_type;
   pair(pair const&) = default;
   pair() = default;
   pair& operator=(pair const&) = default;
};
```

More difficult is the case of one void argument. You still need a template parameter for the other part of the pair. That calls for *partial specialization*.

Partial Specialization

When you write a template specialization that involves some, but not all, of the template arguments, it is called *partial specialization*. Some programmers call explicit specialization *full specialization* to help distinguish it from partial specialization. Partial specialization is explicit, so the phrase full specialization is more descriptive, and I will use it for the rest of this book.

Begin a partial specialization with a template header that lists the template parameters you are not specializing. Then define the specialization. As with full specialization, name the class that you are specializing by listing all the template arguments. Some of the template arguments depend on the specialization's parameters, and some are fixed with specific values. That's what makes this specialization partial.

As with full specialization, the definition of the specialization completely replaces the primary template for a particular set of template arguments. By convention, you keep the same interface, but the actual implementation is up to you.

Listing 51-3 shows a partial specialization of pair if the first template argument is void.

Listing 51-3. Specializing `pair` for One `void` Argument

```
template<class U>
struct pair<void, U>
{
   typedef void first_type;
   typedef U second_type;
   U second;
   pair() = default;
   pair(pair const&) = default;
   pair(U const& second);
   template<class U2>
   pair(pair<void, U2> const& other);
};
```

Based on Listing 51-3, **write a partial specialization of pair with a void second argument.** Compare your solution with Listing 51-4.

Listing 51-4. Specializing `pair` for the Other `void` Argument

```
template<class T>
struct pair<T, void>
{
   typedef T first_type;
   typedef void second_type;
   T first;
   pair() = default;
   pair(pair const&) = default;
   pair(T const& first);
   template<class T2>
   pair(pair<T2, void> const& other);
};
```

Regardless of the presence or absence of any partial or full specializations, you still use the pair template the same way: always with two type arguments. The compiler examines those template arguments and determines which specialization to use.

Partially Specializing Function Templates

You cannot partially specialize a function template. Full specialization is allowed, as described in the previous Exploration, but partial specialization is not. Sorry. Use overloading instead, which is usually better than template specialization anyway.

Value Template Parameters

Before I present the next example of partial specialization, I want to introduce a new template feature. Templates typically use types as parameters, but they can also use values. Declare a value template parameter with a type and optional name, much the same way that you would declare a function parameter. Value template parameters are limited to types for which you can specify a compile-time constant: bool, char, int, etc., but floating-point types, string literals, and classes are not allowed.

For example, suppose you want to modify the fixed class that you wrote for Exploration 47 so that the developer can specify the number of digits after the decimal place. While you're at it, you can also use a template parameter to specify the underlying type, as shown in Listing 51-5.

Listing 51-5. Changing `fixed` from a Class to a Class Template

```
template<class T, int N>
class fixed
{
public:
    typedef T value_type;
    static constexpr int places{N};
    static constexpr int places10{power10(N)};
    fixed();
    fixed(T const& integer, T const& fraction);
    fixed& operator=(fixed const& rhs);
    fixed& operator+=(fixed const& rhs);
    fixed& operator*=(fixed const& rhs);
    // and so on...
private:
    T value_; // scaled to N decimal places
};

template<class T, int N>
bool operator==(fixed<T,N> const& a, fixed<T,N> const& b);
template<class T, int N>
fixed<T,N> operator+(fixed<T,N> const& a, fixed<T,N> const& b);
template<class T, int N>
fixed<T,N> operator*(fixed<T,N> const& a, fixed<T,N> const& b);
// ... and so on...
```

The key challenge in converting the fixed class to a class template is defining places10 in terms of places. C++ has no exponentiation operator, but you can write a constexpr function to compute a simple power of 10. (Another possible solution is to use template trickery, but such an advanced topic will have to wait until Exploration 70.) Recall from the previous Exploration that a constexpr function must have only one statement: a return statement. Another limitation is that you cannot declare any local variables. A common technique, well-known in functional-programming circles, is to use a helper function that takes an additional argument to store the computation in progress. Call the helper function recursively. Here is a situation in which the conditional operator is necessary to halt the recursion. See Listing 51-6 for the power10 function and its helper.

Listing 51-6. Computing a Power of 10 at Compile Time

```
/// Called from power10 to compute 10<sup>@p n</sup>, storing the result so far in @p result.
int constexpr power10_helper(int n, int result)
{
  return n == 0 ? result : power10_helper(n - 1, 10 * result);
}

/// Compute a power of 10 at compile time. The type T must support subtraction and multiplication.
int constexpr power10(int n)
{
  return power10_helper(n, 1);
}
```

Suppose you have an application that instantiates fixed<long, 0>. This degenerate case is no different from a plain long, but with overhead and complexity for managing the implicit decimal point. Suppose further that performance measurements of your application reveal that this overhead has a measurable impact on the overall performance of the application. Therefore, you decide to use partial specialization for the case of fixed<T, 0>. Use a partial specialization, so that the template still takes a template argument for the underlying type.

You might wonder why the application programmer doesn't simply replace fixed<long, 0> with plain long. In some cases, that is the correct solution. Other times, however, the use of fixed<long, 0> might be buried inside another template. The issue, therefore, becomes one of which template to specialize. For the sake of this Exploration, we are specializing fixed.

Remember that any specialization must provide a full implementation. You don't have to specialize the free functions too. By specializing the fixed class template, we get the performance boost we need. Listing 51-7 shows the partial specialization of fixed.

Listing 51-7. Specializing `fixed` for `N == 0`

```
template<class T>
class fixed<T, 0>
{
public:
    typedef T value_type;
    static constexpr T places{0};
    static constexpr T places10{1};
    fixed() : value_{} {}
    fixed(T const& integer, T const&) : value_{integer} {}
    fixed& operator=(fixed const& rhs) { value_ = rhs; }
    fixed& operator+=(fixed const& rhs) { value_ += rhs; }
    fixed& operator*=(fixed const& rhs) { value_ *= rhs; }
    ... and so on...
private:
    T value_; // no need for scaling
};
```

The next Exploration introduces a language feature that helps you manage your custom types: namespaces.

EXPLORATION 52

Names and Namespaces

Nearly every name in the standard library begins with std::, and only names in the standard library are permitted to start with std::. For your own names, you can define other prefixes, which is a good idea and an excellent way to avoid name collisions. Libraries and large programs in particular benefit from proper partitioning and naming. However, templates and names have some complications, and this Exploration helps clarify the issues.

Namespaces

The name std is an example of a *namespace*, which is a C++ term for a named scope. A namespace is a way to keep names organized. When you see a name that begins with std::, you know it's in the standard library. Good third-party libraries use namespaces. The open-source Boost project (www.boost.org), for example, uses the boost namespace to ensure names, such as boost::container::vector, do not interfere with similar names in the standard library, such as std::vector. Applications can take advantage of namespaces too. For example, different project teams can place their own names in different namespaces, so members of one team are free to name functions and classes without the need to check with other teams. For example, the GUI team might use the namespace gui and define a gui::tree class, which manages a tree widget in a user interface. The database team might use the db namespace. Thus, db::tree might represent a tree data structure that is used to store database indexes on disk. A database debugging tool can use both tree classes, because there is no clash between db::tree and gui::tree. The namespaces keep the names separate.

To create a namespace and declare names within it, you must define the namespace. A namespace definition begins with the namespace keyword, followed by an optional identifier that names the namespace. This, in turn, is followed by declarations within curly braces. Unlike a class definition, a namespace definition does not end with a semicolon after the closing curly brace. All the declarations within the curly braces are in the scope of the namespace. You must define a namespace outside of any function. Listing 52-1 defines the namespace numeric and, within it, the rational class template.

Listing 52-1. Defining the `rational` Class Template in the `numeric` Namespace

```
#ifndef RATIONAL_HPP_
#define RATIONAL_HPP_

namespace numeric
{
  template<class T>
  class rational
  {
    ... you know what goes here...
  };
  template<class T>
  bool operator==(rational<T> const& a, rational<T> const& b);
  template<class T>
  rational<T> operator+(rational<T> const& a, rational<T> const& b);
  ... and so on...
} // namespace numeric

#endif
```

Namespace definitions can be discontiguous. This means that you can have many separate namespace blocks that all contribute to the same namespace. Therefore, multiple header files can each define the same namespace, and every definition adds names to the same, common namespace. Listing 52-2 illustrates how to define the fixed class template within the same numeric namespace, even in a different header (say, fixed.hpp).

Listing 52-2. Defining the `fixed` Class Template in the `numeric` Namespace

```
#ifndef FIXED_HPP_
#define FIXED_HPP_

namespace numeric
{
  template<class T, int N>
  class fixed
  {
    ... copied from Exploration 51...
  };
  template<class T, int N>
  bool operator==(fixed<T,N> const& a, fixed<T,N> const& b);
  template<class T, int N>
  fixed<T,N> operator+(fixed<T,N> const& a, fixed<T,N> const& b);
  // and so on...
} // namespace numeric

#endif
```

Note how the free functions and operators that are associated with the class templates are defined in the same namespace. I'll explain exactly why later in the Exploration, but I wanted to point it out now, because it's very important.

When you declare but don't define an entity (such as a function) in a namespace, you have a choice for how to define that entity, as described in the following:

- Use the same or another namespace definition and define the entity within the namespace definition.
- Define the entity outside of the namespace and prefix the entity name with the namespace name and the scope operator (::).

Listing 52-3 illustrates both styles of definition. (The declarations are in Listings 52-1 and 52-2.)

Listing 52-3. Defining Entities in a Namespace

```
namespace numeric
{
  template<class T>
  rational<T> operator+(rational<T> const& a, rational<T> const& b)
  {
    rational<T> result{a};
    result += b;
    return result;
  }
}

template<class T, int N>
numeric::fixed<T, N> numeric::operator+(fixed<T, N> const& a, fixed<T, N> const& b)
{
  fixed<T, N> result{a};
  result += b;
  return result;
}
```

The first form is straightforward. As always, the definition must follow the declaration. In a header file, you might define an inline function or function template using this syntax.

In the second form, the compiler sees the namespace name (numeric), followed by the scope operator, and knows to look up the subsequent name (operator*) in that namespace. The compiler considers the rest of the function to be in the namespace scope, so you don't have to specify the namespace name in the remainder of the declaration (that is, the function parameters and the function body). The function's return type comes before the function name, which places it outside the namespace scope, so you still have to use the namespace name. To avoid ambiguity, you are not allowed to have a namespace and a class with the same name in a single namespace.

The alternative style of writing a function return type that I touched on in Exploration 23 lets you write the return type without repeating the namespace scope, because the function name establishes the scope for you, as shown in Listing 52-4.

Listing 52-4. Alternative Style of Function Declaration in a Namespace

```
template<class T, int N>
auto numeric::operator+(fixed<T, N> const& a, fixed<T, N> const& b) -> fixed<T, N>
{
  fixed<T, N> result{a};
  result += b;
  return result;
}
```

Traditionally, when you define a namespace in a header, the header contains a single namespace definition, which contains all the necessary declarations and definitions. When you implement functions and other entities in a separate source file, I find it most convenient to write an explicit namespace and define the functions inside the namespace, but some programmers prefer to omit the namespace definition. Instead, they use the namespace name and scope operator when defining the entities. An entity name that begins with the namespace name and scope operator is an example of a *qualified* name—that is, a name that explicitly tells the compiler where to look to find the name's declaration.

The name rational<int>::value_type is qualified, because the compiler knows to look up value_type in the class template rational, specialized for int. The name std::vector is a qualified name, because the compiler looks up vector in the namespace std. On the other hand, where does the compiler look up the name std? Before I can answer that question, I have to delve into the general subject of nested namespaces.

Nested Namespaces

Namespaces can nest, that is, you can define a namespace inside another namespace, as demonstrated in the following:

```
namespace exploring_cpp
{
  namespace numeric {
    template<class T> class rational
    {
      // and so on ...
    };
  }
}
```

To use a nested namespace, the qualifier lists all the namespaces in order, starting from the outermost namespace. Separate each namespace with the scope operator (::).

```
exploring_cpp::numeric::rational<int> half{1, 2};
```

A top-level namespace, such as std or exploring_cpp, is actually a nested namespace. Its outer namespace is called the *global namespace*. All entities that you declare outside of any function are in a namespace—in an explicit namespace or in the global namespace. Thus, names outside of functions are said to be at *namespace scope*. The phrase *global scope* refers to names that are declared in the implicit global namespace, which means outside of any explicit namespace. Qualify a global name by prefixing the name with a scope operator.

```
::exploring_cpp::numeric::rational<int> half{1, 2};
```

Most programs you read will not use an explicit global scope operator. Instead, programmers tend to rely on the normal C++ rules for looking up names, letting the compiler find global names on its own. So far, every function you've written has been global; every call to these functions has been unqualified. The compiler has never had a problem with the unqualified names. If you have a situation in which a local name hides a global name, you can refer to the global name explicitly. Listing 52-5 demonstrates the kind of trouble you can wreak through poor choice of names and how to use qualified names to extricate yourself.

Listing 52-5. Coping with Conflicting Names

```
#include <cmath>
#include <numeric>
#include <vector>

namespace stats {
  // Really bad name for a functor to compute sum of squares,
  // for use in determining standard deviation.
  class std
  {
  public:
    std(double mean) : mean_{mean} {}
    double operator()(double acc, double x)
    const
14     {
15       return acc + square(x - mean_);
16     }
17     double square(double x) const { return x * x; }
18   private:
19     double mean_;
20   };
21
22   // Really bad name for a function in the stats namespace.
23   // It computes standard deviation.
24   double stats(::std::vector<double> const& data)
25   {
26     double std{0.0}; // Really, really bad name for a local variable
27     if (not data.empty())
28     {
29       double sum{::std::accumulate(data.begin(), data.end(), 0.0)};
30       double mean{sum / data.size()};
31       double sumsq{::std::accumulate(data.begin(), data.end(), 0.0,
32                    stats::std(mean))};
33       double variance{sumsq / data.size() - mean * mean};
34       std = ::std::sqrt(variance);
35     }
36     return std;
37   }
38 }
```

The local variable std does not conflict with the namespace of the same name because the compiler knows that only class and namespace names can appear on the left-hand side of a scope operator. On the other hand, the class std does conflict, so the use of a bare std:: qualifier is ambiguous. You must use ::std (for the standard library namespace) or stats::std (for the class). References to the local variable must use a plain std.

The name stats on line 24 names a function, so it does not conflict with the namespace stats. Therefore, the use of stats::std on line 32 is not ambiguous.

The accumulate algorithm, called on lines 29 and 31, does exactly what its name suggests. It adds all the elements in a range to a starting value, either by invoking the + operator or by calling a binary functor that takes the sum and a value from the range as arguments.

Remove the global scope operators from ::std::accumulate (lines 29 and 31) to give std::accumulate. Recompile the program. **What messages does your compiler give you?**

Restore the file to its original form. **Remove the first :: qualifier from ::std::vector (line 24). What message does the compiler give you?**

Restore the file to its original form. **Remove the stats:: qualifier from stats::std (line 32). What message does the compiler give you?**

____

____

____

Sane and rational people do not deliberately name a class std in a C++ program, but we all make mistakes. (Maybe you have a class that represents a building element in an architectural CAD system, and you accidentally omitted the letter u from stud.) By seeing the kinds of messages that the compiler issues when it runs into name conflicts, you can better recognize these errors when you accidentally create a name that conflicts with a name invented by a third-party library or another team working on your project.

Most application programmers don't have to use the global scope prefix, because you can be careful about choosing names that don't conflict. Library authors, on the other hand, never know where their code will be used or what names that code will use. Therefore, cautious library authors often use the global scope prefix.

Global Namespace

Names that you declare outside of all namespaces are *global*. In the past, I used *global* to mean "outside of any function," but that was before you knew about namespaces. C++ programmers refer to names declared at *namespace scope*, which is our way of saying, "outside of any function." Such a name can be declared in a namespace or outside of any explicit namespace.

A program's main function must be global, that is, at namespace scope, but not in any namespace. If you define another function named main in a namespace, it does not interfere with the global main, but it will confuse anyone who reads your program.

The std Namespace

As you know, the standard library uses the std namespace. You are not allowed to define any names in the std namespace, but you can specialize templates that are defined in std, provided at least one template argument is a user-defined type.

The C++ standard library inherits some functions, types, and objects from the C standard library. You can recognize the C-derived headers, because their names begin with an extra letter c; e.g., <cmath> is the C++ equivalent of the C header <math.h>. Some C names, such as EOF, do not follow namespace rules. These names are usually written in all capital letters, to warn you that they are special. You don't have to concern yourself with the details; just be aware that you cannot use the scope operator with these names, and the names are always global. When you look up a name in a language reference, these special names are called *macros*. (You've already seen one example: assert, declared in the <cassert> header, even though it is not all capital letters.)

The C++ standard grants some flexibility in how a library implementation inherits the C standard library. The specific rule is that a C header of the form <header.h> (for some C headers, such as math) declares its names in the global namespace, and the implementation decides whether the names are also in the std namespace. The C header with the form <cheader> declares its names in the std namespace, and the implementation may also declare them in the global namespace. Regardless of the style you choose, all C standard functions are reserved to the implementation, which means you are not free to use any C standard function name in the global namespace. If you want to use the same name, you must declare it in a different namespace. Some people like <cstddef> and std::size_t, and others prefer <stddef.h> and size_t. Pick a style and stick with it.

My recommendation is not to get caught up in which names originate in the C standard library and which are unique to C++. Instead, consider any name in the standard library off-limits. The only exception is when you want to use the same name for the same purpose, but in your own namespace. For example, you may want to overload the abs function to work with rational or fixed objects. Do so in their respective namespaces, alongside all the overloaded operators and other free functions.

■ **Caution** Many C++ references omit the C portions of the standard library. As you can see, however, the C portions are most problematic when it comes to name clashes. Thus, make sure your C++ 11 reference is complete or supplement your incomplete C++ reference with a complete C 99 library reference.

Using Namespaces

In order to use any name, the C++ compiler must be able to find it, which means identifying the scope in which it is declared. The most direct way to use a name from a namespace, such as rational or fixed, is to use a qualified name—that is, the namespace name as a prefix, e.g., numeric, followed by the scope operator(::).

```
numeric::rational<long> pi_r{80143857L, 25510582L};
numeric::fixed<long, 6> pi_f{3, 141593};
```

When the compiler sees the namespace name and the double colons (::), it knows to look up the subsequent name in that namespace. There is no chance of a collision with the same entity name in a different namespace.

Sometimes, however, you end up using the namespace a lot, and brevity becomes a virtue. The next two sections describe a couple of options.

The using Directive

You've seen a using directive before, but in case you need a refresher, take a look at the following:

```
using namespace std;
```

The syntax is as follows: the using keyword, the namespace keyword, and a namespace name. A using directive instructs the compiler to treat all the names in the namespace as though they were global. (The precise rule is slightly more complicated. However, unless you have a nested hierarchy of namespaces, the simplification is accurate.) You can list multiple using directives, but you run the risk of introducing name collisions among the namespaces. A using directive affects only the scope in which you place it. Because it can have a big impact on name lookup, restrict using directives to the narrowest scope you can; typically this would be an inner block.

Although a using directive has its advantages—and I use them in this book—you must be careful. They hinder the key advantage of namespaces: avoidance of name collisions. Names in different namespaces don't ordinarily collide, but if you try to mix namespaces that declare a common name, the compiler will complain.

If you are careless with using directives, you can accidentally use a name from the wrong namespace. If you're lucky, the compiler will tell you about your mistake, because your code uses the wrong name in a way that violates language rules. If you aren't lucky, the wrong name will coincidentally have the same syntax, and you won't notice your mistake until much, much later.

Never place a using directive in a header. That ruins namespaces for everyone who includes your header. Keep using directives as local as possible, in the smallest scope possible.

In general, I try to avoid using directives. You should get used to reading fully qualified names. On the other hand, sometimes long names interfere with easy comprehension of complicated code. Rarely do I use more than one using directive in the same scope. So far, the only times I've ever done so is when all the namespaces are defined by the same library, so I know they work together, and I won't run into naming problems. Listing 52-6 illustrates how using directives work.

Listing 52-6. Examples of using Directives

```
#include <iostream>

void print(int i)
{
  std::cout << "int: " << i << '\n';
}

namespace labeled
{
  void print(double d)
  {
    std::cout << "double: " << d << '\n';
  }
}

namespace simple
{
  void print(int i)
  {
    std::cout << i << '\n';
  }
  void print(double d)
  {
    std::cout << d << '\n';
  }
}

void test_simple()
{
  using namespace simple;
  print(42);                // ???
  print(3.14159);           // finds simple::print(double)
}

void test_labeled()
{
  using namespace labeled;
  print(42);                // find ::print(int)
  print(3.14159);           // finds labeled::print(double)
}

int main()
{
  test_simple();
  test_labeled();
}
```

What will happen if you try to compile Listing 52-5?

The error is on line 31. The using directive effectively merges the simple namespace with the global namespace. Thus, you now have two functions named print that take a single int argument, and the compiler doesn't know which one you want. Fix the problem by qualifying the call to print(42) (on line 32), so it calls the function in the simple namespace. **What do you expect as the program output?**

Try it. Make sure you get what you expect. Line 31 should now look like the following:

```
simple::print(42);
```

The using Declaration

More specific, and therefore less dangerous, than a using directive is a using declaration. A using declaration imports a single name from another namespace into a local scope, as demonstrated in the following:

```
using numeric::rational;
```

A using declaration adds the name to the local scope as though you had declared it explicitly. Thus, within the scope where you place the using declaration, you can use the declared name without qualification (e.g., rational). Listing 52-7 shows how using declarations help avoid the problems you encountered with *using* directives in Listing 52-6.

Listing 52-7. Examples of `using` Declarations with Namespaces

```
#include <iostream>

void print(int i)
{
  std::cout << "int: " << i << '\n';
}

namespace labeled
{
  void print(double d)
  {
    std::cout << "double: " << d << '\n';
  }
}

```

```
16 namespace simple
17 {
18   void print(int i)
19   {
20     std::cout << i << '\n';
21  }
22   void print(double d)
23   {
24     std::cout << d << '\n';
25   }
26 }
27
28 void test_simple()
29 {
30   using simple::print;
31   print(42);
32   print(3.14159);
33 }
34
35 void test_labeled()
36 {
37   using labeled::print;
38   print(42);
39   print(3.14159);
40 }
41
42 int main()
43 {
44   test_simple();
45   test_labeled();
46 }
```

Predict the program's output.

This time, the compiler can find simple::print(int), because the using declaration injects names into the local scope. Thus, the local names do not conflict with the global print(int) function. On the other hand, the compiler does not call ::print(int) for line 38. Instead, it calls labeled::print(double), converting 42 to 42.0.

Are you puzzled by the compiler's behavior? Let me explain. When the compiler tries to resolve an overloaded function or operator name, it looks for the first scope that declares a matching name. It then collects all overloaded names from that scope, and only from that scope. Finally, it resolves the name by choosing the best match (or reporting an error, if it cannot find exactly one good match). Once the compiler finds a match, it stops looking in other scopes or outer namespaces.

In this case, the compiler sees the call to print(42) and looks for the name print first in the local scope, where it finds a function named print that was imported from the labeled namespace. So it stops looking for namespaces and tries to resolve the name print. It finds one function, which takes a double argument. The compiler knows how to convert an int to a double, so it deems this function a match and calls it. The compiler never even looks at the global namespace.

How would you instruct the compiler to also consider the global print function?

____

Add a using declaration for the global print function. Between lines 37 and 38, insert the following:

```
using ::print;
```

When the compiler tries to resolve print(int), it finds labeled::print(double) and ::print(int), both imported into the local scope. It then resolves the overload by considering both functions. The print(int) function is the best match for an int argument.

Now add using simple::print; at the same location. **What do you expect to happen when you compile this example now?**

____

Now the compiler has too many choices—and they conflict. A using directive doesn't cause this kind of conflict, because it simply changes the namespaces where the compiler looks for a name. A using declaration, however, adds a declaration to the local scope. If you add too many declarations, those declarations can conflict, and the compiler would complain.

When a using declaration names a template, the template name is brought into the local scope. The compiler keeps track of full and partial specializations of a template. The using declaration affects only whether the compiler finds the template at all. Once it finds the template and decides to instantiate it, the compiler will find the proper specialization. That's why you can specialize a template that is defined in the standard library—that is, in the std namespace.

A key difference between a using directive and a using declaration is that a using directive does not affect the local scope. A using declaration, however, introduces the unqualified name into the local scope. This means you cannot declare your own name in the same scope. Listing 52-8 illustrates the difference.

Listing 52-8. Comparing a `using` Directive with a `using` Declaration

```
#include <iostream>

void demonstrate_using_directive()
{
   using namespace std;
   typedef int ostream;
   ostream x{0};
   std::cout << x << '\n';
}

void demonstrate_using_declaration()
{
   using std::ostream;
   typedef int ostream;
   ostream x{0};
   std::cout << x << '\n';
}
```

The local declaration of ostream interferes with the using declaration, but not the using directive. A local scope can have only one object or type with a particular name, and a using declaration adds the name to the local scope, whereas a using directive does not.

The using Declaration in a Class

A using declaration can also import a member of a class. This is different from a namespace *using* declaration, because you can't just import any old member into any old class, but you can import a name from a base class into a derived class. There are several reasons why you may want to do this. Two immediate reasons are:

- The base class declares a function, and the derived class declares a function with the same name, and you want overloading to find both functions. The compiler looks for overloads only in a single class scope. With a using declaration to import the base class function into the derived class scope, overloading can find both functions in the derived class scope and so choose the best match.
- When inheriting privately, you can selectively expose members by placing a using declaration in a public section of the derived class.

Listing 52-9 illustrates using declarations. You will learn more advantages of using declarations as you learn more advanced C++ techniques.

Listing 52-9. Examples of `using` Declarations with Classes

```
#include <iostream>

class base
{
public:
  void print(int i) { std::cout << "base: " << i << '\n'; }
};

class derived1 : public base
{
public:
  void print(double d) { std::cout << "derived: " << d << '\n'; }
};

class derived2 : public base
{
public:
  using base::print;
  void print(double d) { std::cout << "derived: " << d << '\n'; }
};

int main()
{
  derived1 d1{};
  derived2 d2{};

  d1.print(42);
  d2.print(42);
}
```

Predict the output from the program.

____

____

The class derived1 has a single member function named print. Calling d1.print(42) converts 42 to 42.0 and calls that function. Class derived2 imports print from the base class. Thus, overloading determines the best match for d2.print(42) and calls print in the base class. The output appears as follows:

```
derived: 42
base: 42
```

Unnamed Namespaces

A name is optional in a namespace definition. The names in an ordinary, named namespace are shared among all files that make up a program, but names in an unnamed namespace are private to the source file that contains the namespace definition.

```
namespace {
  // Version control ID string is different in every file.
  const std::string id("$Id$");
}
```

When you want to keep various helper functions and other implementation details private, define them in an unnamed namespace (sometimes called an *anonymous* namespace). This ensures that their names will not collide with the same names in any other source files. (Note to C programmers: Use anonymous namespaces instead of global static functions and objects.) Restrict your use of the anonymous namespace to source files, not headers. If a header file contains an unnamed namespace, every source file that includes the header is furnished with its own private copy of everything defined in that namespace. Sometimes you want everyone to have a separate, anonymous copy, but usually not.

The only tricky aspect of the unnamed namespace is that you cannot qualify names to refer to names that you defined in the anonymous namespace. You must rely on ordinary name lookup for unqualified names. The next section discusses name lookup issues in greater depth.

Name Lookup

In the absence of namespaces, looking up a function or operator name is simple. The compiler looks in the local block first, then in the outer blocks and the inner namespace before the outer namespace, until, finally, the compiler searches global declarations. It stops searching in the first block that contains a matching declaration. If the compiler is looking for a function or operator, the name may be overloaded, so the compiler considers all the matching names that are declared in the same scope, regardless of parameters.

Looking up a member function is slightly different. When the compiler looks up an unqualified name in a class context, it starts by searching in the local block and enclosing blocks, as described earlier. The search continues by considering members of the class, then its base class, and so on for all ancestor classes. Again, when looking up an overloaded name, the compiler considers all the matching names that it finds in the same scope—that is, the same class or block.

Namespaces complicate the name lookup rules. Suppose you want to use the rational type, which is defined in the exploring_cpp::numeric namespace. You know how to use a qualified name for the type, but what about, for instance, addition or the I/O operators, such as those in the following:

```
exploring_cpp::numeric::rational<int> r;
std::cout << r + 1 << '\n';
```

The full name of the addition operator is exploring_cpp::numeric::operator+. But normally, you use the addition operator without specifying the namespace. Therefore, the compiler needs some help to determine which namespace contains the operator declaration. The trick is that the compiler checks the types of the operands and looks for the overloaded operator in the namespaces that contain those types. This is known as argument-dependent lookup (ADL). It is also called Koenig Lookup, after Andrew Koenig, who first described ADL.

The compiler collects several sets of scopes to search. It first determines which scopes to search using the ordinary lookup rules, described at the beginning of this section. For each function argument or operator operand, the compiler also collects a set of namespaces based on the argument types. If a type is a class type, the compiler selects the namespace that contains the class declaration and the namespaces that contain all of its ancestor classes. If the type is a specialization of a class template, the compiler selects the namespace that contains the primary template and the namespaces of all the template arguments. The compiler forms the union of all these scopes and then searches them for the function or operator. As you can see, the goal of ADL is to be inclusive. The compiler tries hard to discover which scope declares the operator or function name.

To better understand the importance of ADL, take a look at Listing 52-10.

Listing 52-10. Reading and Writing Tokens

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

namespace parser
{
  class token
  {
  public:
    token() : text_{} {}
    token(std::string& s) : text_{s} {}
    token& operator=(std::string const& s) { text_ = s; return *this; }
    std::string text() const { return text_; }
  private:
    std::string text_;
  };
}

std::istream& operator>>(std::istream& in, parser::token& tok)
{
  std::string str{};
  if (in >> str)
    tok = str;
  return in;
}
```

```
std::ostream& operator<<(std::ostream& out, parser::token const& tok)
{
  out << tok.text();
  return out;
}

int main()
{
  using namespace parser;
  using namespace std;

  vector<token> tokens{};
  copy(istream_iterator<token>(std::cin), istream_iterator<token>(),
       back_inserter(tokens));
  copy(tokens.begin(), tokens.end(), ostream_iterator<token>(cout, "\n"));
}
```

What will happen when you compile the program?

____

____

Some compilers, trying to be helpful, fill your console with messages. The core of the problem is that istream_iterator and ostream_iterator invoke the standard input (>>) and output (<<) operators. In the case of Listing 52-10, the compiler locates the operators through ordinary lookup as member functions of the istream and ostream classes. The standard library declares these member function operators for the built-in types, so the compiler cannot find a match for an argument of type parser::token. Because the compiler finds a match in a class scope, it never gets around to searching the global scope, so it never finds the custom I/O operators.

The compiler applies ADL and searches the parser namespace because the second operand to << and >> has type parser::token. It searches the std namespace because the first operand has type std::istream or std::ostream. It cannot find a match for the I/O operators in these namespaces, because the operators are in the global scope.

Now you see why it's vital that you declare all associated operators in the same namespace as the main type. If you don't, the compiler cannot find them. **Move the I/O operators into the parser namespace and see that the program now works.** Compare your program with Listing 52-11.

Listing 52-11. Move the I/O Operators into the `parser` Namespace

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

namespace parser
{
  class token
  {
  public:
    token() : text_{} {}
    token(std::string& s) : text_{s} {}
    token& operator=(std::string const& s) { text_ = s; return *this; }
    std::string text() const { return text_; }
  private:
    std::string text_;
  };

  std::istream& operator>>(std::istream& in, parser::token& tok)
  {
    std::string str{};
    if (in >> str)
      tok = str;
    return in;
  }

  std::ostream& operator<<(std::ostream& out, parser::token const& tok)
  {
    out << tok.text();
    return out;
  }
}

int main()
{
  using namespace parser;
  using namespace std;

  vector<token> tokens{};
  copy(istream_iterator<token>(std::cin), istream_iterator<token>(),
       back_inserter(tokens));
  copy(tokens.begin(), tokens.end(), ostream_iterator<token>(cout, "\n"));
}
```

To see how the compiler extends its ADL search, **modify the program to change the container from a vector to a map, and count the number of occurrences of each token.** (Remember Exploration 22?) Because a map stores pair objects, write an output operator that prints pairs of tokens and counts. This means ostream_iterator calls the << operator with two arguments from namespace std. Nonetheless, the compiler finds your operator (in the parser namespace), because the template argument to std::pair is in parser. Your program may end up looking something like Listing 52-12.

Listing 52-12. Counting Occurrences of Tokens

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <map>
#include <string>

namespace parser
{
  class token
  {
  public:
    token() : text_{} {}
    token(std::string& s) : text_{s} {}
    token& operator=(std::string const& s) { text_ = s; return *this; }
    std::string text() const { return text_; }
  private:
    std::string text_;
  };

  // To store tokens in a map.
  bool operator<(token const& a, token const& b)
  {
    return a.text() < b.text();
  }

  std::istream& operator>>(std::istream& in, parser::token& tok)
  {
    std::string str{};
    if (in >> str)
      tok = str;
    return in;
  }

  std::ostream& operator<<(std::ostream& out, parser::token const& tok)
  {
    out << tok.text();
    return out;
  }

  std::ostream& operator<<(std::ostream& out,
                           std::pair<const token, long> const& count)
  {
    out << count.first.text() << '\t' << count.second << '\n';
    return out;
  }
}

int main()
{
  using namespace parser;
  using namespace std;

  map<token, long> tokens{};
  token tok{};
  while (std::cin >> tok)
     ++tokens[tok];
  copy(tokens.begin(), tokens.end(),
       ostream_iterator<pair<const token, long> >(cout));
}
```

Now that you know about templates and namespaces, it's time to look at some of their practical uses. The next several Explorations take a closer look at parts of the standard library, beginning with the standard containers.

EXPLORATION 53

Containers

So far, the only standard containers you've used have been vector and map. I mentioned array in Exploration 9 and list in Exploration 44 but never went into depth. This Exploration introduces the remaining containers and discusses the general nature of containers. When third-party libraries implement additional containers, they usually follow the pattern set by the standard library and make their containers follow the same requirements.

Properties of Containers

The container types implement familiar data structures, such as trees, lists, arrays, and so on. They all serve the common purpose of storing a collection of similar objects in a single container object. You can treat the container as a single entity: compare it, copy it, assign it, and so on. You can also access the individual items in the container. What distinguishes one container type from another is how the container stores the items within it, which in turn affects the speed of accessing and modifying items in the container.

The standard containers fall into two broad categories: sequence and associative. The difference is that you can control the order of items in a sequence container but not in an associative container. As a result, associative containers offer improved performance for accessing and modifying their contents. The standard sequence containers are array (fixed-size), deque (double-ended queue), forward_list (singly linked list), list (doubly linked list), and vector (variable-length array). The forward_list type works differently from the other containers (due to the nature of singly linked lists) and is for specialized uses. This book does not cover forward_list, but you can find it in any C++ 11 reference.

The associative containers have two subcategories: ordered and unordered. Ordered containers store keys in a data-dependent order, which is given by the < operator or a caller-supplied functor. Although the standard does not specify any particular implementation, the complexity requirements pretty much dictate that ordered associative containers are implemented as balanced binary trees. Unordered containers store keys in a hash table, so the order is unimportant to your code and is subject to change as you add items to the container.

Another way to divide the associative containers is into sets and maps. Sets are like mathematical sets: they have members and can test for membership. Maps are like sets that store key/value pairs. Sets and maps can require unique keys or permit duplicate keys. The set types are set (unique key, ordered), multiset (duplicate key, ordered), unordered_set, and unordered_multiset. The map types are map, multimap, unordered_map, and unordered_multimap.

Different containers have different characteristics. For example, vector permits rapid access to any item, but insertion in the middle is slow. A list, on the other hand, offers rapid insertion and erasure of any item but provides only bidirectional iterators, not random access. The unordered containers do not permit comparing entire containers.

The C++ standard defines container characteristics in terms of *complexity*, which is written in big-O notation. Remember from your introductory algorithms course that $O(1)$ is constant complexity, but without any indication of what the constant might be. $O(n)$ is linear complexity: if the container has n items, performing an $O(n)$ operation takes time proportional to n. Operations on sorted data are often logarithmic: $O(\log n)$.

Table 53-1 summarizes all the containers and their characteristics. The Insert, Erase, and Lookup columns show the average-case complexity for these operations, where N is the number of elements in the container. Lookup for a sequence container means looking for an item at a particular index. For an associative container, it means looking for a specific item by value. "No" means the container does not support that operation at all.

Table 53-1. *Summary of Containers and Their Characteristics*

Type	Header	Insert	Erase	Lookup	Iterator
array	<array>	No	No	*O*(1)	Random Access
deque	<deque>	*O*(*N*)*	*O*(*N*)*	*O*(1)	Random Access
forward_list	<forward_list>	*O*(1)	*O*(1)	*O*(*N*)	Forward
list	<list>	*O*(1)	*O*(1)	*O*(*N*)	Bidirectional
map	<map>	*O*(log *N*)	*O*(log *N*)	*O*(log *N*)	Bidirectional
multimap	<map>	*O*(log *N*)	*O*(log *N*)	*O*(log *N*)	Bidirectional
multiset	<set>	*O*(log *N*)	*O*(log *N*)	*O*(log *N*)	Bidirectional
set	<set>	*O*(log *N*)	*O*(log *N*)	*O*(log *N*)	Bidirectional
unordered_map	<unordered_map>	*O*(1)	*O*(1)	*O*(1)	Forward
unordered_multimap	<unordered_map>	*O*(1)	*O*(1)	*O*(1)	Forward
unordered_multiset	<unordered_set>	*O*(1)	*O*(1)	*O*(1)	Forward
unordered_set	<unordered_set>	*O*(1)	*O*(1)	*O*(1)	Forward
vector	<vector>	*O*(*N*)*	*O*(*N*)*	*O*(1)	Random Access

*Complexity is O(N) for insertion and erasure in the middle of the container but is O(1) at the end of the container, when amortized over many operations. A deque also allows amortized O(1) insertion and erasure at the beginning of the container.

Member Types

Every container provides a number of useful types and typedefs as members of the container. This section makes frequent use of several of them:

value_type

This is a synonym for the type that the container stores. For example, value_type for vector<double> is double and std::list<char>::value_type is char. Using a standard typedef makes it easier to write and read container code. The rest of this Exploration uses value_type extensively.

The mapped containers store key/value pairs, so value_type for map<Key, T> (and multimap, unordered_map, and unordered_multimap) is std::pair<const Key, T>. The key type is const, because you cannot change the key after adding an item to an associative container. The internal structure of the container depends on the keys, so changing the keys would violate the ordering constraints.

key_type

The associative containers declare key_type as a typedef for the first template parameter—for example, map<int, double>::key_type is int. For the set types, key_type and value_type are the same.

reference

This is a synonym for a reference to value_type. Except in very rare cases, reference is identical to value_type&.

const_reference

const_reference is a synonym for a reference to const value_type. Except in very rare cases, const_reference is identical to value_type const&.

iterator

This is the iterator type. It might be a typedef, but more likely, it is a class, the definition of which is implementation-dependent. All that matters is that this type meets the requirements of an iterator. Each container type implements a specific category of iterator, as described in Table 53-1.

const_iterator

const_iterator is the iterator type for const items. It might be a typedef, but more likely, it is a class, the definition of which is implementation-dependent. All that matters is that this type meets the requirements of an iterator of const items. Each container type implements a specific category of iterator, as described in Table 53-1.

size_type

size_type is a typedef for one of the built-in integer types (which one depends on the implementation). It represents an index for a sequence container or a container size.

What Can Go into a Container

In order to store an item in a container, the item's type must meet some basic requirements. You must be able to copy or move the item and assign it using copy or move. For the built-in types, this is automatic. For a class type, you usually have that capability. The compiler even writes the constructors and assignment operators for you. So far, all the classes in this book meet the basic requirements; you won't have to concern yourself with non-conforming classes until Exploration 59.

Sequence containers themselves do not have to compare items for equality; they just copy or move elements as required. When they have to make copies, they assume that copies are identical to the original.

Ordered associative containers require an ordering functor. By default, they use a standard functor called std::less<key_type>, which in turn uses the < operator. You can supply a custom functor, provided it implements *strict weak ordering*, which is defined by the following requirements:

- If $a < b$ and $b < c$, then $a < c$.
- $a < a$ is always false.
- The order does not change after an item is stored in a container.

A common error among new C++ programmers is to violate rule 2, typically by implementing <= instead of <. Violations of the strict weak ordering rule result in undefined behavior. Some libraries have a debugging mode that checks your functor to ensure that it is valid. If your library has such a mode, use it.

Unordered associative containers need a hash functor and an equality functor. The default hash functor is std::hash<key_type> (declared in <functional>). The standard library provides specializations for the built-in types and string. If you store a custom class in an unordered container, you have to provide your own hash functor. The simplest way to do that is to specialize hash. Listing 53-1 shows how to specialize hash for the rational type. You have to provide only the function call operator, which must return type std::size_t (an implementation-defined integer type).

Listing 53-1. Specializing the `hash` Template for the `rational` Type

```
#include <array>
#include "rational.hpp"
namespace std {

template<class T>
class hash<rational<T>>
{
public:
  std::size_t operator()(rational<T> const& r)
  const
  {
    return hasher_(r.numerator()) + hasher_(r.denominator());
  }
private:
  std::hash<T> hasher_;
};
} // end of std
```

The default equality functor is std::equal_to<T> (declared in <functional>), which uses the == operator. If two items are equal, their hash values must also be equal (but the reverse is not necessarily true).

When you insert an item in a container, the container keeps a copy of the item, or you can move an object into a container. When you erase an item, the container destroys the item. When you destroy a container, it destroys all of its elements. The next section discusses insertion and erasure at greater length.

Inserting and Erasing

I've presented a number of examples of inserting and erasing elements in vectors and maps. This section explores this topic in greater depth. Except for array and forward_list, the container types follow some basic patterns. The array type has a fixed size, so it provides none of the insertion or erasure functions. And forward_list has its own way of doing things, because a singly linked list cannot insert or erase items directly. All the other containers follow the specification described in this section.

Inserting in a Sequence Container

You have a choice of several member functions with which to insert an item into a sequence container. The most fundamental function is emplace, which constructs an item in place in the container. It has the following form:

iterator emplace(iterator here, args...)

Inserts a newly constructed item into the collection immediately before the position here and returns an iterator that refers to the newly added item. The args can be zero or more arguments that are passed to the value_type constructor. If here is end(), the item is constructed at the end of the container.

If you have objects that are already constructed, call insert, which has three overloaded forms:

> **`iterator insert(iterator here, value_type item)`**
>
> Inserts item by copying or moving it into the collection immediately before the position here and returns an iterator that refers to the newly added item. If here is end(), item is appended to the end of the container.
>
> **`void insert(iterator here, size_type n, value_type const& item)`**
>
> Inserts n copies of item immediately before the position to which here refers. If here is end(), the items are appended to the end of the container.
>
> **`template<class InputIterator>void insert(iterator here, InputIterator first, InputIterator last)`**
>
> Copies the values from the range (first, last) into the container, starting at the position immediately before here.

Two additional functions add items to the start (emplace_front and push_front), and two add items to the end (emplace_back and push_back) of a container. The emplace functions pass all their arguments to the constructor of the new element; the push functions copy or move their sole argument into the container. The container type provides only the functions that it can implement with constant complexity. Thus, vector provides emplace_back and push_back but not emplace_front or push_front. Both list and deque provide all four functions.

Erasing from a Sequence Container

The erase function erases, or deletes, items from a container. Sequence containers implement two forms of erase:

> **`iterator erase(iterator pos)`**
>
> Erases the item to which pos refers and returns an iterator that refers to the subsequent item. Returns end() if the last item is erased. The behavior is undefined if you try to erase end() or if pos is an iterator for a different container object.
>
> **`iterator erase(iterator first, iterator last)`**
>
> Erases all the items in the range [first, last) and returns an iterator that refers to the item that immediately followed the last item erased. Returns end() if the last item in the container is erased. The behavior is undefined if the iterators are in the wrong order or refer to a different container object.

The clear() function erases all elements from the container. In addition to the basic erasure functions, sequence containers also provide pop_front to erase the first element and pop_back to erase the last element of a collection. A container implements these two functions only if it can do so with constant complexity. **Which sequence containers implement pop_back?**

____

Which sequence containers implement pop_front?

____

As with the push functions, vector provides pop_back, and list and deque provide both pop_back and pop_front.

Inserting in an Associative Container

All the insertion functions for associative containers follow a common pattern for return types. The duplicate-key containers (multimap, multiset, unordered_multimap, unordered_multiset) return an iterator for the newly added item. The unique-key containers (map, set, unordered_map, unordered_set) return a pair<iterator, bool>: the iterator refers to the item in the container, and the bool is true if the item was added or false if the item was already present. In this section, the return type is shown as *return*. If the item was already present, the existing item is untouched, and the new item is ignored.

Construct a new item in an associative container by calling one of the two emplace functions:

return emplace(args...)

Constructs a new element at the correct position in the container, passing args to the value_type constructor.

iterator emplace_hint(iterator hint, args...)

Constructs a new element as close to hint as possible, passing args to the value_type constructor.

For an ordered container, if the item's position is immediately after hint, the item is added with constant complexity. Otherwise, the complexity is logarithmic. If you have to store many items in an ordered container, and the items are already in order, you can save some time by using the position of the most recently inserted item as the hint.

As with sequence containers, you can also call the insert function to insert into an associative container. One key difference from the sequence containers is that you don't have to provide a position (one form does let you provide a position as a hint).

return **insert(value_type item)**

Moves or copies item into the container.

iterator insert(iterator hint, value_type item)

Moves or copies item into the container as close to hint as possible, as described earlier with emplace_hint.**template<class InputIterator>void insert(InputIterator first, InputIterator last)**

Copies the values from the range [first, last) into the container. For the ordered containers, you get optimal performance if the range [first, last) is already sorted.

Write a program that reads a list of strings from the standard input into a set of strings. Use the emplace_hint function. Save the return value to pass as the hint when inserting the next item. Find a large list of strings to use as input. Make two copies of the list, one in sorted order and one in random order. (See this book's web site if you need help locating or preparing the input files.) **Compare the performance of your program reading the two input files.**

Write another version of the same program, this time using the simple, one-argument emplace **function**. Again, run the program with both input files. Compare the performance of all four variations: hinted and unhinted insert, sorted and unsorted input.

Listing 53-2 shows a simple form of the program that uses emplace_hint.

Listing 53-2. Using a Hint Position When Inserting into a Set

```
#include <iostream>
#include <set>
#include <string>

int main()
{
  std::set<std::string> words{};

  std::set<std::string>::iterator hint{words.begin()};
  std::string word{};
  while(std::cin >> word)
    hint = words.emplace_hint(hint, std::move(word));
}
```

When I run the program with a file of more than 200,000 words, the hinted program with sorted input executes in about 1.6 seconds. The unhinted form takes 2.2 seconds. With randomly ordered inputs, both programs run in about 2.3 seconds. As you can see, the hint can make a difference when the input is already sorted. The details depend on the library implementation; your mileage may vary.

Erasing from an Associative Container

The erase function erases, or deletes, items from a container. Associative containers implement two forms of erase:

`void erase(iterator pos)iterator erase(iterator pos)`

Erases the item to which pos refers; complexity is constant, possibly amortized over many calls. The ordered containers do not return a value; unordered containers return an iterator that refers to the successor value (or end()). The behavior is undefined if you try to erase end() or if pos is an iterator for a different container object.

`void erase(iterator first, iterator last)iterator erase(iterator first, iterator last)`

Erases all the items in the range [first, last). Ordered containers do not return a value; unordered containers return an iterator that refers to the item that follows the last item erased. Returns end( ) if the last item in the container is erased. The behavior is undefined if the iterators are in the wrong order or refer to a different container object.

As with sequence containers, clear( ) erases all elements of the container.

Exceptions

The containers do their best to keep order if an exception is thrown. Exceptions have two potential sources: the container itself and the items in the containers. Most member functions do not throw exceptions for invalid arguments, so the most common source of exceptions from the container itself is std::bad_alloc, if the container runs out of memory and cannot insert a new item.

If you try to insert a single item into a container, and the operation fails (perhaps because the item's copy constructor throws an exception, or the container ran out of memory), the container is unchanged.

If you try to insert multiple items, and one of those items throws an exception while it is being inserted into a container (e.g., the item's copy constructor throws an exception), most containers do not roll back the change. Only the list and forward_list types roll back to their original state. The other containers leave the container in a valid state, and the items that have been inserted successfully remain in the container.

When erasing one or many items, the containers do not throw exceptions themselves, but they may have to move (or in the case of ordered containers, compare) items; if an item's move constructor throws an exception (a highly unlikely event), the erasure may be incomplete. No matter what, however, the container remains in a valid state.

In order for these guarantees to remain valid, destructors must not throw exceptions.

■ **Tip** Never throw an exception from a destructor.

Iterators and References

When using containers, one important point that I have not yet covered is the validity of iterators and references. The issue is that when you insert or erase items in a container, some or all iterators for that container can become invalid, and references to items in the container can become invalid. The details of which iterators and references become invalid and under what circumstances depend on the container.

Iterators and references becoming invalid reflect the internal structure of the container. For example, vector stores its elements in a single, contiguous chunk of memory. Therefore, inserting or erasing any elements shifts all the elements at higher indices, which invalidates all iterators and references to those elements at higher indices. As a vector grows, it may have to allocate a new internal array, which invalidates all extant iterators and references for that vector. You never know when that can occur, so it is safest never to hold onto a vector's iterators or references while adding items to the vector. (But look up the reserve member function in a library reference, if you must keep those iterators and references lying around.)

A list, on the other hand, implements a doubly linked list. Inserting or erasing an element is simply a matter of inserting or deleting a node, which has no effect on iterators and references to other nodes. For all containers, if you erase a node that an iterator refers to, that iterator necessarily becomes invalid, just as a reference to the erased element must become invalid.

In practical terms, you must take care when inserting and erasing elements. These functions often return iterators that you can use to help maintain your program's logic. Listing 53-3 shows a function template, erase_less, which marches through a container and calls erase for any element that is less than the value that precedes it. It is a function template, and it works with any class that meets the requirements of a sequence container.

Listing 53-3. Erasing Elements from a Sequence Container

```
template<class Container>
void erase_less(Container& cont)
{
  typename Container::iterator prev{cont.end()};
  typename Container::iterator iter{cont.begin()};
  while (iter != cont.end())
  {
    if (prev != cont.end() and not (*prev < *iter))
      iter = cont.erase(iter);
    else
    {
      prev = iter;
      ++iter;
    }
  }
}
```

Notice how erase_less moves the iterator, iter, through the container. The prev iterator refers to the previous item (or container.end(), when the loop first begins and there is no previous item). As long as *prev is less than *iter, the loop advances by setting prev to iter and incrementing iter. If the container is in ascending order, nothing happens to it. If an item is out of place, however, *prev < *iter is false, and the item at position iter is erased. The value that erase returns is an iterator that refers to the item that follows iter prior to its erasure. That's exactly where we want iter to point, so we just set iter to the return value and let the loop continue.

The typename keyword tells the compiler that iterator is the name of a type. Because Container is a template, the compiler does not know what the iterator member is (object? function? typedef?), until the template is instantiated. But first it has to parse the template definition, and to do that, it must know what kind of name iterator is. The rule is that the compiler assumes the name is an expression, unless you tell it that the name is a type, by using the typename keyword.

Write a test program to see that erase_less works with a list and with a vector. Make sure it works with ascending data, descending data, and mixed data. Listing 53-4 shows my simple test program.

Listing 53-4. Testing the `erase_less` Function Template

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <iterator>
#include <list>
#include <sstream>
#include <vector>

#include "erase_less.hpp" //Listing 53-3

/// Extract items from a string and store them in a container.
template<class Container>
void read(std::string const& str, Container& cont)
{
  std::istringstream in{str};
  cont.insert(cont.begin(),
              std::istream_iterator<typename Container::value_type>(in),
              std::istream_iterator<typename Container::value_type>());
}

/// Print items from a container to the standard output.
template<class Container>
void print(std::string const& label, Container const& cont)
{
  std::cout << label;
  std::copy(cont.begin(), cont.end(),
            std::ostream_iterator<typename Container::value_type>(std::cout, " "));
  std::cout << '\n';
}
```

```
/// Test erase_less by extracting integers from a string into a container
/// and calling erase_less. Print the container before and after.
/// Double-check that the same results obtain with a list and a vector.
void test(std::string const& str)
{
  std::list<int> list{};
  read(str, list);
  print("before: ", list);
  erase_less(list);
  print("after:  ", list);

  std::vector<int> vector{};
  read(str, vector);
  erase_less(vector);

  assert(list.size() == vector.size());
  assert(std::equal(list.begin(), list.end(), vector.begin()));
}

int main()
{
  test("2 3 7 11 13 17 23 29 31 37");
  test("37 31 29 23 17 13 11 7 3 2");
  test("");
  test("42");
  test("10 30 20 40 0 50");
}
```

Again, notice the use of the typename keyword to inform the compiler that what follows is the name of a type. Use typename only in a template and only for names that the compiler cannot determine on its own, such as members of a template parameter or members of a template that uses a template parameter as an argument.

Sequence Containers

In this book, the most common use of a container has been to add items to the end of vector. A program might then use standard algorithms to change the order, such as sorting into ascending order, shuffling into random order, etc. In addition to vector, the other sequence containers are array, deque, and list.

The primary distinguishing feature of the sequence containers is their complexity characteristics. If you often have to insert and erase from the middle of the sequence, you probably want list. If you have to insert and erase only off one end, use vector. If the container's size is a fixed, compile-time constant, use array. If the elements of the sequence must be stored contiguously (in a single block of memory), use array or vector.

The following sections include some more details about each container type. Each section presents the same program for comparison. The program constructs a deck of playing cards then randomly selects a card for itself and a card for you. The card with the highest value wins. The program plays ten times then exits. The program plays without replacement—that is, it does not return used cards to the deck after each game. In order to pick a random card, the programs use the randomint class, from Listing 43-5. Save the class definition in a file named randomint.hpp or download the file from the book's web site. Listing 53-5 shows the card class, which the sample programs use. For the full class definition, download card.cpp from the book's web site.

Listing 53-5. The card Class, to Represent a Playing Card

```
#ifndef CARD_HPP_
#define CARD_HPP_

#include <iosfwd>

/// Represent a standard western playing card.
class card
{
public:
  typedef char suit;
  static suit const spades   {4};
  static suit const hearts   {3};
  static suit const clubs    {2};
  static suit const diamonds {1};

  typedef char rank;
  static rank const ace   {14};
  static rank const king  {13};
  static rank const queen {12};
  static rank const jack  {11};

  card() : rank_{0}, suit_{0} {}
  card(rank r, suit s) : rank_{r}, suit_{s} {}

  void assign(rank r, suit s);
  suit get_suit() const { return suit_; }
  rank get_rank() const { return rank_; }
private:
  rank rank_;
  suit suit_;
};

bool operator==(card a, card b);
bool operator!=(card a, card b);
std::ostream& operator<<(std::ostream& out, card c);
std::istream& operator>>(std::istream& in, card& c);

/// In some games, Aces are high. In other Aces are low. Use different
/// comparison functors depending on the game.
bool acehigh_compare(card a, card b);
bool acelow_compare(card a, card b);

/// Generate successive playing cards, in a well-defined order,
/// namely, 2-10, J, Q, K, A. Diamonds first, then Clubs, Hearts, and Spades.
/// Roll-over and start at the beginning again after generating 52 cards.
class card_generator
{
public:
  card_generator();
  card operator()();
private:
  card card_;
};

#endif
```

The array Class Template

The array type is a fixed-size container, so you cannot call insert or erase. To use array, specify a base type and a size as a compile-time constant expression, as shown in the following:

```
std::array<double, 5> five_elements;
```

If you initialize an array with fewer values than the array size, remaining values are initialized to zero. If you omit the initializer altogether, the compiler calls the default initializer if the value type is a class type; otherwise it leaves the array elements uninitialized. Because an array cannot change size, you can't simply erase the cards after playing. In order to keep the code simple, the program returns cards to the deck after each game. Listing 53-6 shows the high-card program with replacement.

Listing 53-6. Playing High-Card with array

```
#include <array>
#include <iostream>

#include "card.hpp"
#include "randomint.hpp" // Listing 43-5

int main()
{
  std::array<card, 52> deck;
  std::generate(deck.begin(), deck.end(), card_generator{});

  randomint picker{0, deck.size() - 1};
  for (int i{0}; i != 10; ++i)
  {
    card const& computer_card{deck.at(picker())};
    std::cout << "I picked " << computer_card << '\n';

    card const& user_card{deck.at(picker())};
    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";
  }
}
```

The deque Class Template

A deque (pronounced “deck”) represents a double-ended queue. Insertion and erasure from the beginning or the end is fast, but the complexity is linear, if you have to insert or erase anywhere else. Most of the time, you can use deque the same way you would use avector, so **apply your experience with vector to write the high-card program.** Play without replacement: that is, after each game, discard the two cards by erasing them from the container. Listing 53-7 shows how I wrote the high-card program using a deque.

Listing 53-7. Playing High-Card with a deque

```
#include <deque>
#include <iostream>

#include "card.hpp"
#include "randomint.hpp"

int main()
{
  std::deque<card> deck(52);
  std::generate(deck.begin(), deck.end(), card_generator{});

  for (int i{0}; i != 10; ++i)
  {
    std::deque<card>::iterator pick{deck.begin() + randomint{0, deck.size()-1}()};
    card computer_card{*pick};
    deck.erase(pick);

    std::cout << "I picked " << computer_card << '\n';

    pick = deck.begin() + randomint{0, deck.size() - 1}();
    card user_card{*pick};
    deck.erase(pick);

    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";

  }
}
```

The list Class Template

A list represents a doubly linked list. Insertion and erasure is fast at any point in the list, but random access is not supported. Thus, the high-card program uses iterators and the advance function (Exploration 44). **Write the high-card program to use list.** Compare your solution with that of mine in Listing 53-8.

Listing 53-8. Playing High-Card with a `list`

```
#include <iostream>
#include <list>

#include "card.hpp"
#include "randomint.hpp"

int main()
{
  std::list<card> deck(52);
  std::generate(deck.begin(), deck.end(), card_generator{});

  for (int i{0}; i != 10; ++i)
  {
    std::list<card>::iterator pick{deck.begin()};
    std::advance(pick, randomint{0, deck.size() - 1}());
    card computer_card{*pick};
    deck.erase(pick);

    std::cout << "I picked " << computer_card << '\n';

    pick = std::next(deck.begin(), randomint{0, deck.size() - 1}());

    card user_card{*pick};

    deck.erase(pick);

    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";

  }
}
```

The deque type supports random access iterators, so it could add an integer to begin( ) to pick a card. But list uses bidirectional iterators, so it must call advance( ) or next( ); Listing 53-8 demonstrates both. Note that you can call advance() or next() for deques, too, and the implementation would still use addition.

The vector Class Template

A vector is an array that can change size at runtime. Appending to the end or erasing from the end is fast, but complexity is linear when inserting or erasing anywhere else in the vector. **Compare deque and list versions of the high-card program. Pick the one you prefer and modify it to work with vector.** My version of the program is displayed in Listing 53-9.

Listing 53-9. Playing High-Card with `vector`

```
#include <iostream>
#include <vector>

#include "card.hpp"
#include "randomint.hpp"

int main()
{
  std::vector<card> deck(52);
  std::generate(deck.begin(), deck.end(), card_generator{});

  for (int i{0}; i != 10; ++i)
  {
    std::vector<card>::iterator pick{deck.begin() + randomint{0, deck.size()-1}()};
    card computer_card{*pick};
    deck.erase(pick);

    std::cout << "I picked " << computer_card << '\n';

    pick = deck.begin() + randomint{0, deck.size() - 1}();
    card user_card{*pick};
    deck.erase(pick);

    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";

  }
}
```

Notice how you can change the program to use vector instead of deque, just by changing the type names. Their usage is quite similar. One key difference is that deque offers fast (constant complexity) insertion at the beginning of the container, something that vector lacks. The other key difference is that vector stores all of its elements in a single chunk of memory, which can be important when interfacing with external libraries. Neither of these factors matters here.

Associative Containers

The associative containers offer rapid insertion, deletion, and lookup, by controlling the order of elements in the container. The ordered associative containers store elements in a tree, ordered by a comparison functor (default is std::less, which uses <), so insertion, erasure, and lookup occur with logarithmic complexity. The unordered containers use hash tables (according to a caller-supplied hash functor and equality functor) for access with constant complexity in the average case, but with a linear worst-case complexity. Consult any textbook on data structures and algorithms for more information regarding trees and hash tables.

Sets store keys, and maps store key/value pairs. Multisets and multimaps allow duplicate keys. All equivalent keys are stored at adjacent locations in the container. Plain sets and maps require unique keys. If you try to insert a key that is already in the container, the new key is not inserted. Remember that equivalence in an ordered container is determined solely by calling the comparison functor: compare(a, b) is false, and compare(b, a) is false means a and b are equivalent. Unordered containers call their equality functor to determine whether a key is a duplicate. The default is std::equal_to (declared in <functional>), which uses the == operator.

Because associative arrays store keys in an order that depends on the keys' contents, you cannot modify the contents of a key that is stored in an associative container. This means you cannot use an associative container's iterators as output iterators. Thus, if you want to implement the high-card program using an associative container, you can use the inserter function to create an output iterator that fills the container. Listing 53-10 shows how to use set to implement the high-card program.

Listing 53-10. Playing High-Card with `set`

```
#include <iostream>
#include <iterator>
#include <set>
#include <utility>

#include "card.hpp"
#include "randomint.hpp"

int main()
{
  typedef std::set<card, std::function<bool(card, card)>> cardset;
  cardset deck(acehigh_compare);
  std::generate_n(std::inserter(deck, deck.begin()), 52, card_generator{});

  for (int i{0}; i != 10; ++i)
  {
    cardset::iterator pick{deck.begin()};
    std::advance(pick, randomint{0, deck.size() - 1}());
    card computer_card{*pick};
    deck.erase(pick);

    std::cout << "I picked " << computer_card << '\n';

    pick = deck.begin();
    std::advance(pick, randomint{0, deck.size() - 1}());
    card user_card{*pick};
    deck.erase(pick);

    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";

  }
}
```

When using associative containers, you may experience some difficulty when you use a custom compare functor (for ordered containers) or custom equality and hash functors (for unordered containers). You must specify the functor type as a template argument. When you construct a container object, pass a functor as an argument to the constructor. The functor must be an instance of the type that you specified in the template specialization.

For example, Listing 53-10 uses the acehigh_compare function, which it passes to the constructor for deck. Because acehigh_compare is a function, you must specify a function type as the template argument. The easiest way to declare a function type is with the std::function template. The template argument looks somewhat like a nameless function header: supply the return type and parameter types:

```
std::function<bool(card, card)>
```

Another approach is to specialize the std::less class template for type card. The explicit specialization would implement the function call operator to call acehigh_compare. Taking advantage of the specialization, you could use the default template argument and constructor arguments. The specialization should declare some standard member typedefs. Follow the pattern of functors in the <functional> header. The functor should provide a function call operator that uses the argument and return types and implements the strict weak ordering function for your container. Listing 53-11 demonstrates yet another version of the high-card program, this time using a specialization of less. The only real difference is how the deck is initialized.

Listing 53-11. Playing High-Card Using an Explicit Specialization of `std::less`

```
#include <functional>
#include <iostream>
#include <iterator>
#include <set>

#include "card.hpp"
#include "randomint.hpp"

namespace std
{
  template<>
  class less<card>
  {
  public:
    typedef card first_argument_type;
    typedef card second_argument_type;
    typedef bool result_type;
    bool operator()(card a, card b) const { return acehigh_compare(a, b); }
  };
}

int main()
{
  typedef std::set<card> cardset;
  cardset deck{};
  std::generate_n(std::inserter(deck, deck.begin()), 52, card_generator{});
  for (int i{0}; i != 10; ++i)
  {
    cardset::iterator pick{deck.begin()};
    std::advance(pick, randomint{0, deck.size() - 1}());
    card computer_card{*pick};
    deck.erase(pick);

    std::cout << "I picked " << computer_card << '\n';

    pick = deck.begin();
    std::advance(pick, randomint{0, deck.size() - 1}());
    card user_card{*pick};
    deck.erase(pick);

    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";

  }
}
```

To use an unordered container, you must write an explicit specialization of std::hash<card>. Listing 53-1 should be able to help. The *card.hpp* header already declares operator== for card, so you should be ready to **rewrite the high-card program one last time, this time for unordered_set.** Compare your solution with Listing 53-12. All these programs are remarkably similar, in spite of the different container types, a feat made possible through the judicious use of iterators, algorithms, and functors.

Listing 53-12. Playing High-Card with `unordered_set`

```
#include <functional>
#include <iostream>
#include <iterator>
#include <unordered_set>

#include "card.hpp"
#include "randomint.hpp"

namespace std
{
  template<>
  class hash<card>
  {
  public:
    std::size_t operator()(card a)
    const
    {
      return hash<card::suit>{}(a.get_suit()) * hash<card::rank>{}(a.get_rank());
    }
  };
} // namespace std

int main()
{
  typedef std::unordered_set<card> cardset;
  cardset deck{};
  std::generate_n(std::inserter(deck, deck.begin()), 52, card_generator{});

  for (int i(0); i != 10; ++i)
  {
    cardset::iterator pick{deck.begin()};
    std::advance(pick, randomint{0, deck.size() - 1}());
    card computer_card{*pick};
    deck.erase(pick);

    std::cout << "I picked " << computer_card << '\n';

    pick = deck.begin();
    std::advance(pick, randomint{0, deck.size() - 1}());
    card user_card{*pick};
    deck.erase(pick);

    std::cout << "You picked " << user_card << '\n';

    if (acehigh_compare(computer_card, user_card))
      std::cout << "You win.\n";
    else
      std::cout << "I win.\n";

  }
}
```

In the next Exploration, you will embark on a completely different journey, one involving world travels to exotic locations, where natives speak exotic languages and use exotic character sets. The journey also touches on new and interesting uses for templates.

EXPLORATION 54

Locales and Facets

As you saw in Exploration 18, C++ offers a complicated system to support internationalization and localization of your code. Even if you don't intend to ship translations of your program in a multitude of languages, you must understand the locale mechanism that C++ uses. Indeed, you have been using it all along, because C++ always sends formatted I/O through the locale system. This Exploration will help you understand locales better and make more effective use of them in your programs.

The Problem

The story of the Tower of Babel resonates with programmers. Imagine a world that speaks a single language and uses a single alphabet. How much simpler programming would be if we didn't have to deal with character-set issues, language rules, or locales.

The real world has many languages, numerous alphabets and syllabaries, and multitudinous character sets, all making life far richer and more interesting and making a programmer's job more difficult. Somehow, we programmers must cope. It isn't easy, and this Exploration cannot give you all the answers, but it's a start.

Different cultures, languages, and character sets give rise to different methods to present and interpret information, different interpretations of character codes (as you learned in Exploration 17), and different ways of organizing (especially sorting) information. Even with numeric data, you may find that you have to write the same number in several ways, depending on the local environment, culture, and language. Table 54-1 presents just a few examples of the ways to write a number according to various cultures, conventions, and locales.

Table 54-1. *Various Ways to Write a Number*

Number	Culture
123456.7890	Default C++
123,456.7890	United States
123 456.7890	International scientific
Rs. 1,23,456.7890	Indian currency*
123.456,7890	Germany

**Yes, the commas are correct.*

Other cultural differences can include:

- 12-hour *vs*. 24-hour clock
- How accented characters are sorted relative to non-accented characters (does `'a'` come before or after `'á'`?)
- Date formats: month/day/year, day/month/year, or year-month-day
- Formatting of currency (¥123,456 or 99¢)

Somehow, the poor application programmer must figure out exactly what is culturally-dependent, collect the information for all the possible cultures where the application might run, and use that information appropriately in the application. Fortunately, the hard work has already been done for you and is part of the C++ standard library.

Locales to the Rescue

C++ uses a system called *locales* to manage this disparity of styles. Exploration 18 introduced locales as a means to organize character sets and their properties. Locales also organize formatting of numbers, currency, dates, and times (plus some more stuff that I won't get into).

C++ defines a basic locale, known as the *classic* locale, which provides minimal formatting. Each C++ implementation is then free to provide additional locales. Each locale typically has a name, but the C++ standard does not mandate any particular naming convention, which makes it difficult to write portable code. You can rely on only two standard names:

- The *classic* locale is named `"C"`. The classic locale specifies the same basic formatting information for all implementations. When a program starts, the classic locale is the initial locale.
- An empty string (`""`) means the *default*, or native, locale. The default locale obtains formatting and other information from the host operating system in a manner that depends on what the OS can offer. With traditional desktop operating systems, you can assume that the default locale specifies the user's preferred formatting rules and character-set information. With other environments, such as embedded systems, the default locale may be identical to the classic locale.

A number of C++ implementations use ISO and POSIX standards for naming locales: an ISO 639 code for the language (e.g., `fr` for French, `en` for English, `ko` for Korean), optionally followed by an underscore and an ISO 3166 code for the region (e.g., `CH` for Switzerland, `GB` for Great Britain, `HK` for Hong Kong). The name is optionally followed by a dot and the name of the character set (e.g., `utf8` for Unicode UTF-8, `Big5` for Chinese Big 5 encoding). Thus, I use `en_US.utf8` for my default locale. A native of Taiwan might use `zh_TW.Big5`; developers in French-speaking Switzerland might use `fr_CH.latin1`. Read your library documentation to learn how it specifies locale names. **What is your default locale?** ________________ **What are its main characteristics?**

____

____

____

Every C++ application has a global `locale` object. Unless you explicitly change a stream's locale, it starts off with the global locale. (If you later change the global locale, that does not affect streams that already exist, such as the standard I/O streams.) Initially, the global locale is the classic locale. The classic locale is the same everywhere (except for the parts that depend on the character set), so a program has maximum portability with the classic locale. On the other hand, it has minimum local flavor. The next section explores how you can change a stream's locale.

Locales and I/O

Recall from Exploration 18 that you *imbue* a stream with a locale in order to format I/O according to the locale's rules. Thus, to ensure that you read input in the classic locale, and that you print results in the user's native locale, you need the following:

```
std::cin.imbue(std::locale::classic()); // standard input uses the classic locale
std::cout.imbue(std::locale{""});       // imbue with the user's default locale
```

The standard I/O streams initially use the classic locale. You can imbue a stream with a new locale at any time, but it makes the most sense to do so before performing any I/O.

Typically, you would use the classic locale when reading from, or writing to, files. You usually want the contents of files to be portable and not dependent on a user's OS preferences. For ephemeral output to a console or GUI window, you may want to use the default locale, so the user can be most comfortable reading and understanding it. On the other hand, if there is any chance that another program might try to read your program's output (as happens with UNIX pipes and filters), you should stick with the classic locale, in order to ensure portability and a common format. If you are preparing output to be displayed in a GUI, by all means, use the default locale.

Facets

The way a stream interprets numeric input and formats numeric output is by making requests of the imbued locale. A `locale` object is a collection of pieces, each of which manages a small aspect of internationalization. For example, one piece, called `numpunct`, provides the punctuation symbols for numeric formatting, such as the decimal point character (which is `'.'` in the United States, but `','` in France). Another piece, `num_get`, reads from a stream and parses the text to form a number, using information it obtains from `numpunct`. The pieces such as `num_get` and `numpunct` are called *facets*.

For ordinary numeric I/O, you never have to deal with facets. The I/O streams automatically manage these details for you: the `operator<<` function uses the `num_put` facet to format numbers for output, and `operator>>` uses `num_get` to interpret text as numeric input. For currency, dates, and times, I/O manipulators use facets to format values. But sometimes you need to use facets yourself. The `isalpha`, `toupper`, and other character-related functions about which you learned in Exploration 18 use the `ctype` facet. Any program that has to do a lot of character testing and converting can benefit by managing its facets directly.

Like strings and I/O streams, facets are class templates, parameterized on the character type. So far, the only character type you have used is `char`; you will learn about other character types in Exploration 55. The principles are the same, regardless of character type (which is why facets use templates).

To obtain a facet from a locale, call the `use_facet` function template. The template argument is the facet you seek, and the function argument is the `locale` object. The returned facet is `const` and is not copyable, so the best way to use the result is to initialize a `const` reference, as demonstrated in the following:

```
std::money_get<char> const&
    mgetter{ std::use_facet<std::money_get<char>>(std::locale{""}) };
```

Reading from the inside outward, the object named `mgetter` is initialized to the result of calling the `use_facet` function, which is requesting a reference to the `money_get<char>` facet. The default locale is passed as the sole argument to the `use_facet` function. The type of `mgetter` is a reference to a `const money_get<char>` facet. It's a little daunting to read at first, but you'll get used to it—eventually.

Once you have a facet, call its member functions to use it. This section introduces the currency facets as an example. A complete library reference tells you about all the facets and their member functions.

The money_get facet has two overloaded functions named get. The get function reads a currency value from a sequence of characters specified by an iterator range. It checks the currency symbol, thousands separator, thousands grouping, and decimal point. It extracts the numeric value and stores the value in a double (overloaded form 1) or as a string of digit characters (overloaded form 2). If you choose to use double, take care that you do not run into rounding errors. The get function assumes that the input originates in an input stream, and you must pass a stream object as one of its arguments. It checks the stream's flags to see whether a currency symbol is required (showbase flag is set). And, finally, it sets error flags as needed: failbit for input formatting errors and eofbit for end of file, as follows:

```
std::string digits{};
std::ios_base::iostate error{};
bool international{true};
mgetter.get(std::istreambuf_iterator<char>(stream),  std::istreambuf_iterator<char>(),
   international, stream, error, digits);
```

Similarly, the money_put facet provides the overloaded put function, which formats a double or digit string, according to the locale's currency formatting rules, and writes the formatted value to an output iterator. If the stream's showbase flag is set, money_put prints the currency symbol. The locale's rules specify the position and formatting of the symbol (in the moneypunct facet). The money facets can use local currency rules or international standards. A bool argument to the get and put functions specifies the choice: true for international and false for local. The putter also requires a fill character, as shown in the following example:

```
std::money_put<char> const&
    mputter{ std::use_facet<std::money_put<char>>(std::locale{""}) };
mputter.put(std::ostreambuf_iterator<char>(stream), international, stream, '*', digits);
```

As you can see, using facets directly can be a little complicated. Fortunately, the standard library offers a few I/O manipulators (declared in <iomanip>) to simplify the use of the time and currency facets. Listing 54-1 shows a simple program that imbues the standard I/O streams with the default locale and then reads and writes currency values.

Listing 54-1. Reading and Writing Currency Using the Money I/O Manipulators

```
#include <iomanip>
#include <iostream>
#include <iterator>
#include <locale>
#include <string>

int main()
{
  std::locale native{""};
  std::cin.imbue(native);
  std::cout.imbue(native);

  std::cin >> std::noshowbase;  // currency symbol is optional for input
  std::cout << std::showbase;   // always write the currency symbol for output

  std::string digits;
  while (std::cin >> std::get_money(digits))
  {
    std::cout << std::put_money(digits) << '\n';
  }
  if (not std::cin.eof())
    std::cout << "Invalid input.\n";
}
```

The locale manipulators work like other manipulators, but they invoke the associated facets. The manipulators use the stream to take care of the error flags, iterators, fill character, etc. The `get_time` and `put_time` manipulators read and write dates and times; consult a library reference for details.

Character Categories

This section continues the examination of character sets and locales that you began in Exploration 18. In addition to testing for alphanumeric characters or lowercase characters, you can test for several different categories. Table 54-2 lists all the classification functions and their behavior in the classic locale. They all take a character as the first argument and a `locale` as the second; they all return a `bool` result.

Table 54-2. *Character Classification Functions*

Function	Description	Classic Locale
`isalnum`	Alphanumeric	`'a'-'z'`, `'A'-'Z'`, `'0'-'9'`
`isalpha`	Alphabetic	`'a'-'z'`, `'A'-'Z'`
`iscntrl`	Control	Any non-printable character*
`isdigit`	Digit	`'0'-'9'` (in all locales)
`isgraph`	Graphical	Printable character other than `' '`*
`islower`	Lowercase	`'a'-'z'`
`isprint`	Printable	Any printable character in the character set*
`ispunct`	Punctuation	Printable character other than alphanumeric or white space*
`isspace`	White space	`' '`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`
`isupper`	Uppercase	`'A'-'Z'`
`isxdigit`	Hexadecimal digit	`'a'-'f'`, `'A'-'F'`, `'0'-'9'` (in all locales)

**Behavior depends on the character set, even in the classic locale.*

The classic locale has fixed definitions for some categories (such as `isupper`). Other locales, however, can expand these definitions to include other characters, which may (and probably will) depend on the character set too. Only `isdigit` and `isxdigit` have fixed definitions for all locales and all character sets.

However, even in the classic locale, the precise implementation of some functions, such as `isprint`, depend on the character set. For example, in the popular ISO 8859-1 (Latin-1) character set `'\x80'` is a control character, but in the equally popular Windows-1252 character set, it is printable. In UTF-8, `'\x80'` is invalid, so all the categorization functions would return `false`.

The interaction between the locale and the character set is one of the areas where C++ underperforms. The locale can change at any time, which potentially sets a new character set, which in turn can give new meaning to certain character values. But, the compiler's view of the runtime character set is fixed. For instance, the compiler treats `'A'` as the uppercase Roman letter *A* and compiles the numeric code according to its idea of the runtime character set. That numeric value is then fixed forever. If the characterization functions use the same character set, everything is fine. The `isalpha` and `isupper` functions return true; `isdigit` returns false; and all is right with the world. If the user changes the locale and by so doing changes the character set, those functions may not work with that character variable any more.

Let's consider a concrete example as shown in Listing 54-2. This program encodes locale names, which may not work for your environment. Read the comments and see if your environment can support the same kind of locales, albeit with different names. You will need the `ioflags` class from Listing 39-4. Copy the class to its own header called `ioflags.hpp` or download the file from the book's web site. After reading Listing 54-2, **what do you expect as the result?**

____

____

Listing 54-2. Exploring Character Sets and Locales

```
#include <iomanip>
#include <iostream>
#include <locale>
#include <ostream>

#include "ioflags.hpp"  // from Listing 39-4

/// Print a character's categorization in a locale.
void print(int c, std::string const& name, std::locale loc)
{
  // Don't concern yourself with the & operator. I'll cover that later
  // in the book, in Exploration 63. Its purpose is just to ensure
  // the character's escape code is printed correctly.
  std::cout << "\\x" << std::setw(2) << (c & 0xff) <<
               " is " << name << " in " << loc.name() << '\n';
}

/// Test a character's categorization in the locale, @p loc.
void test(char c, std::locale loc)
{
  ioflags save{std::cout};
  std::cout << std::hex << std::setfill('0');
  if (std::isalnum(c, loc))
    print(c, "alphanumeric", loc);
  else if (std::iscntrl(c, loc))
    print(c, "control", loc);
  else if (std::ispunct(c, loc))
    print(c, "punctuation", loc);
  else
    print(c, "none of the above", loc);
}

int main()
{
  // Test the same code point in different locales and character sets.
  char c{'\xd7'};

  // ISO 8859-1 is also called Latin-1 and is widely used in Western Europe
  // and the Americas. It is often the default character set in these regions.
  // The country and language are unimportant for this test.
  // Choose any that support the ISO 8859-1 character set.
  test(c, std::locale{"en_US.iso88591"});
  // ISO 8859-5 is Cyrillic. It is often the default character set in Russia
  // and some Eastern European countries. Choose any language and region that
  // support the ISO 8859-5 character set.
  test(c, std::locale{"ru_RU.iso88595"});

  // ISO 8859-7 is Greek. Choose any language and region that
  // support the ISO 8859-7 character set.
  test(c, std::locale{"el_GR.iso88597"});

  // ISO 8859-8 contains some Hebrew. The character set is no longer widely used.
  // Choose any language and region that support the ISO 8859-8 character set.
  test(c, std::locale{"he_IL.iso88598"});
}
```

What do you get as the actual response?

____

____

____

____

In case you had trouble identifying locale names or other problems running the program, Listing 54-3 shows the result when I run it on my system.

Listing 54-3. Result of Running the Program in Listing 54-2

```
\xd7 is punctuation in en_US.iso88591
\xd7 is alphanumeric in ru_RU.iso88595
\xd7 is alphanumeric in el_GR.iso88597
\xd7 is none of the above in he_IL.iso88598
```

As you can see, the same character has different categories, depending on the locale's character set. Now imagine that the user has entered a string, and your program has stored the string. If your program changes the global locale or the locale used to process that string, you may end up misinterpreting the string.

In Listing 54-2, the categorization functions reload their facets every time they are called, but you can rewrite the program so it loads its facet only once. The character type facet is called `ctype`. It has a function named `is` that takes a category mask and a character as arguments and returns a `bool`: true if the character has a type in the mask. The mask values are specified in `std::ctype_base`.

■ **Note** Notice the convention that the standard library uses throughout. When a class template needs helper types and constants, they are declared in a non-template base class. The class template derives from the base class and so gains easy access to the types and constants. Callers gain access to the types and constants by qualifying with the base class name. By avoiding the template in the base class, the standard library avoids unnecessary instantiations just to use a type or constant that is unrelated to the template argument.

The mask names are the same as the categorization functions, but without the leading is. Listing 54-4 shows how to rewrite the simple character-set demonstration to use a single cached ctype facet.

Listing 54-4. Caching the ctype Facet

```
#include <iomanip>
#include <iostream>
#include <locale>

#include "ioflags.hpp"  // from Listing 39-4

void print(int c, std::string const& name, std::locale loc)
{
  // Don't concern yourself with the & operator. I'll cover that later
  // in the book. Its purpose is just to ensure the character's escape
  // code is printed correctly.
  std::cout << "\\x" << std::setw(2) << (c & 0xff) <<
               " is " << name << " in " << loc.name() << '\n';
}

/// Test a character's categorization in the locale, @p loc.
void test(char c, std::locale loc)
{
  ioflags save{std::cout};

  std::ctype<char> const& ctype{std::use_facet<std::ctype<char>>(loc)};

  std::cout << std::hex << std::setfill('0');
  if (ctype.is(std::ctype_base::alnum, c))
    print(c, "alphanumeric", loc);
  else if (ctype.is(std::ctype_base::cntrl, c))
    print(c, "control", loc);
  else if (ctype.is(std::ctype_base::punct, c))
    print(c, "punctuation", loc);
  else
    print(c, "none of the above", loc);
}

int main()
{
  // Test the same code point in different locales and character sets.
  char c{'\xd7'};

  // ISO 8859-1 is also called Latin-1 and is widely used in Western Europe
  // and the Americas. It is often the default character set in these regions.
  // The country and language are unimportant for this test.
  // Choose any that support the ISO 8859-1 character set.
  test(c, std::locale{"en_US.iso88591"});

  // ISO 8859-5 is Cyrillic. It is often the default character set in Russia
  // and some Eastern European countries. Choose any language and region that
  // support the ISO 8859-5 character set.
  test(c, std::locale{"ru_RU.iso88595"});
  // ISO 8859-7 is Greek. Choose any language and region that
  // support the ISO 8859-7 character set.
  test(c, std::locale{"el_GR.iso88597"});

  // ISO 8859-8 contains some Hebrew. It is no longer widely used.
  // Choose any language and region that support the ISO 8859-8 character set.
  test(c, std::locale{"he_IL.iso88598"});
}
```

The ctype facet also performs case conversions with the toupper and tolower member functions, which take a single character argument and return a character result. Recall the word-counting problem from Exploration 22. **Rewrite your solution (see Listings 22-2 and 22-3) and change the code to use cached facets.** Compare your program with Listing 54-5.

Listing 54-5. Counting Words Again, This Time with Cached Facets

```
// Copy the initial portion of Listing 22-2 here, including print_counts,
// but stopping just before sanitize.

/** Base class to hold a ctype facet. */
class function
{
public:
  function(std::locale loc) : ctype_{std::use_facet<std::ctype<char>>(loc)} {}
  bool isalnum(char ch) const { return ctype_.is(std::ctype_base::alnum, ch); }
  char tolower(char ch) const { return ctype_.tolower(ch); }
private:
  std::ctype<char> const& ctype_;
};

/** Sanitize a string by keeping only alphabetic characters.
 * @param str the original string
 * @return a santized copy of the string
 */
class sanitizer : public function
{
public:
  typedef std::string argument_type;
  typedef std::string result_type;
  sanitizer(std::locale loc) : function{loc} {}
  std::string operator()(std::string const& str)
  {
    std::string result{};
    for (char c : str)
      if (isalnum(c))
        result.push_back(tolower(c));
    return result;
  }
};

/** Main program to count unique words in the standard input. */
int main()
```

```
{
  // Set the global locale to the native locale.
  std::locale::global(std::locale{""});
  initialize_streams();

  count_map counts{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  std::string word{};
  sanitizer sanitize{std::locale{}};
  while (std::cin >> word)
  {
    std::string copy{sanitize(word)};

    // The "word" might be all punctuation, so the copy would be empty.
    // Don't count empty strings.
    if (not copy.empty())
      ++counts[copy];
  }

  print_counts(counts);
}
```

Notice how most of the program is unchanged. The simple act of caching the `ctype` facet reduces this program's runtime by about 15 percent on my system.

Collation Order

You can use the relational operators (such as `<`) with characters and strings, but they don't actually compare characters or code points; they compare storage units. Most users don't care whether a list of names is sorted in ascending numerical order by storage unit. They want a list of names sorted in ascending alphabetical order, according to their native collation rules.

For example, which comes first: *ångstrom* or *angle*? The answer depends on where you live and what language you speak. In Scandinavia, *angle* comes first, and *ångstrom* follows *zebra*. The `collate` facet compares strings according to the locale's rules. Its `compare` function is somewhat clumsy to use, so the `locale` class template provides a simple interface for determining whether one `string` is less than another in a locale: use the `locale`'s function call operator. In other words, you can use a `locale` object itself as the comparison functor for standard algorithms, such as `sort`. Listing 54-6 shows a program that demonstrates how collation order depends on locale. In order to get the program to run in your environment, you may have to change the locale names.

Listing 54-6. Demonstrating How Collation Order Depends on Locale

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <locale>
#include <string>
#include <vector>
```

```
void sort_words(std::vector<std::string> words, std::locale loc)
{
  std::sort(words.begin(), words.end(), loc);
  std::cout << loc.name() << ":\n";
  std::copy(words.begin(), words.end(),
            std::ostream_iterator<std::string>(std::cout, "\n"));
}

int main()
{
  using namespace std;
  vector<string> words{
    "circus",
    "\u00e5ngstrom",     // ångstrom
    "\u00e7irc\u00ea",   // çircê
    "angle",
    "essen",
    "ether",
    "\u00e6ther",        // æther
    "aether",
    "e\u00dfen"         // eßen
  };
  sort_words(words, locale::classic());
  sort_words(words, locale{"en_GB.utf8"});  // Great Britain
  sort_words(words, locale{"no_NO.utf8"});  // Norway
}
```

The `\uNNNN` characters are a portable way to express Unicode characters. The `NNNN` must be four hexadecimal digits, specifying a Unicode code point. You will learn more in the next Exploration.

The boldface line shows how the `locale` object is used as a comparison functor to sort the words. Table 54-3 lists the results I get for each locale. Depending on your native character set, you may get different results.

Table 54-3. *Collation Order for Each Locale*

Classic	Great Britain	Norway
ångstrom	aether	aether
æther	æther	angle
çircê	angle	çircê
aether	ångstrom	circus
angle	çircê	essen
circus	circus	eßen
eßen	essen	ether
essen	eßen	æther
ether	ether	ångstrom

The next Exploration takes a closer look at international character sets and related difficulties.

EXPLORATION 55

International Characters

Explorations 17–19 discussed characters, but only hinted at bigger things to come. Exploration 54 started to examine these bigger issues with locales and facets. The next topic to explore is how to wrangle character sets and character encodings in an international setting.

This Exploration introduces wide characters, which are like ordinary (or *narrow*) characters, except that they usually occupy more memory. This means the wide character type can potentially represent many more characters than plain `char`. During your exploration of wide characters, you will also get to know more about Unicode.

Why Wide?

As you saw in Exploration 18, the meaning of a particular character value depends on the locale and character set. For instance, in one locale, you can handle Greek characters, while in another locale, Cyrillic, depending on the character set. Your program needs to know the locale and the character set in order to determine which characters are letters, which are punctuation, which are uppercase or lowercase, and how to convert uppercase to lowercase and vice versa.

What if your program has to handle Cyrillic and Greek? What if this program needs to handle them both at the same time? And what about Asian languages? Chinese does not use a Western-style alphabet but instead uses thousands of distinct ideographs. Several Asian languages have adopted some Chinese ideographs for their own use. The typical implementation of the `char` type reaches its limit at only 256 distinct characters, which is woefully inadequate for international demands.

In other words, you can't use plain `char` and `string` types if you want to support the majority of the world's population and their languages. C++ solves this problem with *wide characters*, which it represents using several types: `wchar_t`, `char16_t`, and `char32_t`. (Unlike C's definition of `wchar_t`, the type names in C++ are reserved keywords and built-in types, not typedefs.) The intent is that `wchar_t` is a native type that can represent characters that don't fit into a `char`. With larger characters, a program can support Asian character sets, for example. The `char16_t` and `char32_t` are Unicode types. The Exploration begins by examining `wchar_t`.

Using Wide Characters

In true C++ fashion, the size and other characteristics of `wchar_t` are left to the implementation. The only guarantees are that `wchar_t` is at least as big as `char` and that `wchar_t` is the same size as one of the built-in integer types. The `<cwchar>` header declares a typedef, `std::wint_t`, for that built-in type. In some implementations, `wchar_t` may be identical to `char`, but most desktop and workstation environments use 16 or 32 bits for `wchar_t`.

Dig up Listing 23-2 and modify it to reveal the size of `wchar_t` and `wint_t` in your C++ environment. **How many bits are in wchar_t?** ________________ **How many are in wint_t?** ________________ They should be the same number. **How many bits are in char?** ________________

Wide string objects use the `std::wstring` type (declared in `<string>`). A wide string is a string composed of wide characters. In all other ways, wide strings and narrow strings behave similarly; they have the same member functions, and you use them the same way. For example, the `size()` member function returns the number of characters in the string, regardless of the size of each character.

Wide character and string literals look like their narrow equivalents, except that they start with a capital `L` and they contain wide characters. The best way to express a wide character in a character or string literal is to specify the character's hexadecimal value with the `\x` escape (introduced in Exploration 17). Thus, you have to know the wide character set that your C++ environment uses, and you have to know the numeric value of the desired character in that character set. If your editor and compiler permit it, you may be able to write wide characters directly in a wide character literal, but your source code will not be portable to other environments. You can also write a narrow character in a wide character or string literal, and the compiler automatically converts the narrow characters to wide ones, as shown in the following:

```
wchar_t capital_a{'A'};                    // the compiler automatically widens narrow characters
std::wstring ray{L"Ray"};
wchar_t pi{L'π'};                          // if your tools let you type π as a character
wchar_t pi_unicode{L'\x03c0'};             // if wchar_t uses a Unicode encoding, such as UTF-32
std::wstring price{L"\x20ac" L"12345"};    // Unicode Euro symbol: €12345
```

Notice how in the last line of the example, I divided the string into two parts. Recall from Exploration 17 that the `\x` escape starts an escape sequence that specifies a character by its value in hexadecimal (base 16). The compiler collects as many characters as it can that form a valid hexadecimal number—that is, digits and the letters `A` through `F` (in uppercase or lowercase). It then uses that numeric value as the representation of a single character. If the last line were left as one string, the compiler would try to interpret the entire string as the `\x` escape. This means the compiler would think the character value is the hexadecimal value, $20AC12345_{16}$. By separating the strings, the compiler knows when the `\x` escape ends, and it compiles the character value $20AC_{16}$, followed by the characters `1`, `2`, `3`, `4`, and `5`. Just like narrow strings, the compiler assembles adjacent wide strings into a single wide string. (You are not allowed to place narrow and wide strings next to each other, however. Use all wide strings or all narrow strings, not a mixture of the two.)

Wide Strings

Everything you know about `string` also applies to `wstring`. They are just instances of a common template, `basic_string`. The `<string>` header declares `string` to be a typedef for `basic_string<char>` and `wstring` as a typedef for `basic_string<wchar_t>`. The magic of templates takes care of the details.

Because the underlying implementation of `string` and `wstring` is actually a template, any time you write some utility code to work with strings, you should consider making that code a template too. For example, suppose you want to rewrite the `is_palindrome` function (from Listing 22-5) so that it operates with wide characters. Instead of replacing `char` with `wchar_t`, let's turn it into a function template. Begin by rewriting the supporting functions to be function templates, taking a character type as a template argument. **Rewrite the supporting functions for `is_palindrome` so that they function with narrow and wide strings and characters**. My solution is presented in Listing 55-1.

Listing 55-1. Supporting Cast for the `is_palindrome` Function Template

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <locale>
#include <string>
```

```
/** Test for non-letter.
 * @param ch the character to test
 * @return true if @p ch is not a letter
 */
template<class Char>
bool non_letter(Char ch)
{
  return not std::isalpha(ch, std::locale{});
}

/** Convert to lowercase.
 * Use a canonical form by converting to uppercase first,
 * and then to lowercase.
 * @param ch the character to test
 * @return the character converted to lowercase
 */
template<class Char>
Char lowercase(Char ch)
{
  return std::tolower(ch, std::locale{});
}

/** Compare two characters without regard to case. */
template<class Char>
bool same_char(Char a, Char b)
{
  return lowercase(a) == lowercase(b);
}
```

The next task is to rewrite `is_palindrome` itself. The `basic_string` template actually takes three template arguments. The first is the character type, and the next two are details that needn't concern us at this time. All that matters is that if you want to templatize your own function that deals with strings, you should handle all three of the template parameters.

Before starting, however, you must be aware of a minor hurdle when dealing with functions as arguments to standard algorithms: the argument must be a real function, not the name of a function template. In other words, if you have to work with function templates, such as `lowercase` and `non_letter`, you must instantiate the template and pass the template instance. When you pass `non_letter` and `same_char` to the `remove_if` and `equal` algorithms, be sure to pass the correct template argument too. If `Char` is the template parameter for the character type, use `non_letter<Char>` as the functor argument to `remove_if`.

Rewrite the is_palindrome function as a function template with three template parameters. The first template parameter is the character type: call it `Char`. Call the second template parameter `Traits` and the third `Allocator`. You will have to use all three as arguments to the `std::basic_string` template. Listing 55-2 shows my version of the `is_palindrome` function, converted to a template, so that it can handle narrow and wide strings.

Listing 55-2. Changing `is_palindrome` to a Function Template

```
/** Determine whether @p str is a palindrome.
 * Only letter characters are tested. Spaces and punctuation don't count.
 * Empty strings are not palindromes because that's just too easy.
 * @param str the string to test
 * @return true if @p str is the same forward and backward
 */
```

```
template<class Char, class Traits, class Allocator>
bool is_palindrome(std::basic_string<Char, Traits, Allocator> str)
{
  typedef typename std::basic_string<Char, Traits, Allocator> string;
  typename string::iterator end{
      std::remove_if(str.begin(), str.end(), non_letter<Char>)};
  string rev{str.begin(), end};
  std::reverse(rev.begin(), rev.end());
  return not rev.empty() and
             std::equal(str.begin(), end, rev.begin(), same_char<Char>);
}
```

The `is_palindrome` function never uses the `Traits` or `Allocator` template parameters, except to pass them along to `basic_string`. If you're curious about those parameters, consult a language reference, but be warned that they're a bit advanced. The full name of the `basic_string` specialization is quite long, so a typedef is a great way to turn it into something more manageable. Declaring a local typedef (in this case, `string`) is a common idiom in this situation. Declaring local variables with the `auto` keyword is also common. In a textbook, I prefer to use explicit types.

Calling `is_palindrome` is easy, because the compiler uses automatic type deduction to determine whether you are using narrow or wide strings and instantiates the templates accordingly. Thus, the caller doesn't have to bother with templates at all.

Without further ado, the `non_letter` and `lowercase` functions work with wide character arguments. That's because locales are templates, parameterized on the character type, just like the string and I/O class templates.

However, in order to use wide characters you do have to perform I/O with wide characters, which is the subject of the next section.

Wide Character I/O

You read wide characters from the standard input by reading from `std::wcin`. Write wide characters by writing to `std::wcout` or `std::wcerr`. Once you read or write anything to or from a stream, the character width of the corresponding narrow and wide streams is fixed, and you cannot change it—you must decide whether to use narrow or wide characters and stay with that choice for the lifetime of the stream. So, a program must use `cin` or `wcin`, but not both. Ditto for the output streams. The `<iostream>` header declares the names of all the standard streams, narrow and wide. The `<istream>` header defines all the input stream classes and operators; `<ostream>` defines the output classes and operators. More precisely, `<istream>` and `<ostream>` define templates, and the character type is the first template parameter.

The `<istream>` header defines the `std::basic_istream` class template, parameterized on the character type. The same header declares two typedefs, as follows:

```
typedef basic_istream<char>    istream;
typedef basic_istream<wchar_t> wistream;
```

As you can guess, the `<ostream>` header is similar, defining the `basic_ostream` class template and the `ostream` and `wostream` typedefs.

The `<fstream>` header follows the same pattern—`basic_ifstream` and `basic_ofstream` are class templates, with typedefs, as in the following:

```
typedef basic_ifstream<char>    ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<char>    ofstream;
typedef basic_ofstream<wchar_t> wofstream;
```

Rewrite the main program from Listing 22-5 to test the is_palindrome function template with wide character I/O. Modern desktop environments should be able to support wide characters, but you may have to learn some new features to figure out how to get your text editor to save a file with wide characters. You may also have to load some additional fonts. Most likely, you can supply an ordinary, narrow-text file as input, and the program will work just fine. If you're having difficulty finding a suitable input file, try the palindrome files that you can download with the other examples in this book. The file names indicate the character set. For example, `palindrome-utf8.txt` contains UTF-8 input. You have to determine what format your C++ environment expects when reading a wide stream and pick the correct file. My solution is shown in Listing 55-3.

Listing 55-3. The main Program for Testing `is_palindrome`

```
int main()
{
  std::locale::global(std::locale{""});
  std::wcin.imbue(std::locale{});
  std::wcout.imbue(std::locale{});

  std::wstring line{};
  while (std::getline(std::wcin, line))
    if (is_palindrome(line))
      std::wcout << line << L'\n';
}
```

Reading wide characters from a file or writing wide characters to a file is different from reading or writing narrow characters. All file I/O passes through an additional step of character conversion. C++ always interprets a file as a series of bytes. When reading or writing narrow characters, the conversion of a byte to a narrow character is a no-op, but when reading or writing wide characters, the C++ library has to interpret the bytes to form wide characters. It does so by accumulating one or more adjacent bytes to form each wide character. The rules for deciding which bytes are elements of a wide character and how to combine the characters are specified by the encoding rules for a *multi-byte character set*.

Multi-Byte Character Sets

Multi-byte character sets originated in Asia, where demand for characters exceeded the few character slots available in a single-byte character set, such as ASCII. European nations managed to fit their alphabets into 8-bit character sets, but languages such as Chinese, Japanese, Korean, and Vietnamese require far more bits to represent thousands of ideographs, syllables, and native characters.

The requirements of Asian languages spurred the development of character sets that used two bytes to encode a character—hence the common term, *double-byte character set* (DBCS), with the generalization to *multi-byte character sets* (MBCS). Many DBCSes were invented, and sometimes a single character had multiple encodings. For example, in Chinese Big 5, the ideograph 丁 has the double-byte value `"\xA4\x42"`. In the EUC-KR character set (which is popular in Korea), the same ideograph has a different encoding: `"\xEF\xCB"`.

The typical DBCS uses characters with the most significant bit set (in an 8-bit byte) to represent double characters. Characters with the most significant bit clear would be taken from a single-byte character set (SBCS). Some DBCSes mandate a particular SBCS; others leave it open, so you get different conventions for different combinations of DBCS and SBCS. Mixing single- and double-byte characters in a single character stream is necessary to represent the common use of character streams that mix Asian and Western text. Working with multi-byte characters is more difficult than working with single-byte characters. A string's `size()` function, for example, doesn't tell you how many characters are in a string. You must examine every byte of the string to learn the number of characters. Indexing into a string is more difficult, because you must take care not to index into the middle of a double-byte character.

Sometimes a single character stream needs more flexibility than simply switching between one particular SBCS and one particular DBCS. Sometimes the stream has to mix multiple double-byte character sets. The ISO 2022 standard is an example of a character set that allows shifting between other, subsidiary character sets. *Shift sequences* (also called *escape sequences*, not to be confused with C++ backslash escape sequences) dictate which character set to use. For example, ISO 2022-JP is widely used in Japan and allows switching between ASCII, JIS X 0201 (a SBCS), and JIS X 0208 (a DBCS). Each line of text begins in ASCII, and a shift sequence changes character sets mid-string. For example, the shift sequence `"\x1B$B"` switches to JIS X 0208-1983.

Seeking to an arbitrary position in a file or text stream that contains shift sequences is clearly problematic. A program that has to seek in a multi-byte text stream must keep track of shift sequences in addition to stream positions. Without knowing the most recent shift sequence in the stream, a program has no way of knowing which character set to use to interpret the subsequent characters.

A number of variations on ISO 2022-JP permit additional character sets. The point here is not to offer a tutorial on Asian character sets but to impress on you the complexities of writing a truly open, general, and flexible mechanism that can support the world's rich diversity in character sets and locales. These and similar problems gave rise to the Unicode project.

Unicode

Unicode is an attempt to get out of the whole character-set mess by unifying all major variations into one, big, happy character set. To a large degree, the Unicode Consortium has succeeded. The Unicode character set has been adopted as an international standard as ISO 10646. However, the Unicode project includes more than just the character set; it also specifies rules for case folding, character collation, and more.

Unicode provides 1,114,112 possible character values (called *code points*). So far, the Unicode Consortium has assigned about 100,000 code points to characters, so there's plenty of room for expansion. The simplest way to represent a million code points is to use a 32-bit integer, and indeed, this is a common encoding for Unicode. It is not the only encoding, however. The Unicode standard also defines encodings that let you represent a code point using one or two 16-bit integers and one to four 8-bit integers.

The standard way to write a Unicode code point is U+, followed by the code point as a hexadecimal number of at least four places. Thus, `'\x41'` is the C++ encoding of U+0041 (Latin capital *A*) and Greek π has code point U+03C0. A musical eighth note (♪) has code point U+266A or U+1D160; the former code point is in a group of miscellaneous symbols, which happens to include an eighth note. The latter code point is part of a group of musical symbols, which you will need for any significant work with music-related characters.

UTF-32 is the name of the encoding that stores a code point as a 32-bit integer. To represent a UTF-32 code point in C++, preface the character literal with `U` (uppercase letter *U*). Such a character literal has type `char32_t`. For example, to represent the letter *A*, use `U'A'`; for a lowercase Greek π, use `U'\x03c0'`; and for a musical eighth note (♪), use `U'\x266a'` or `U'\x1d160'`. Do the same for a character string literal, and the standard library defines the type `std::u32string` for a string of `char32_t`. For example, to represent the characters π ≈ 3.14, use the following:

```
std::u32string pi_approx_3_14{ U"\x03c0 \x2248 3.14" };
```

Another common encoding for Unicode uses one to four 8-bit units to make up a single code point. Common characters in Western European languages can usually be represented in a single byte, and many other characters take only two bytes. Less common characters require three or four. The result is an encoding that supports the full range of Unicode code points and almost always consumes less memory than other encodings. This character set is called UTF-8. UTF-8 characters are written as normal character literals, but a UTF-8 string uses the `u8` prefix. The type of a UTF-8 string literal is the same as a normal string literal, but the `u8` prefix tells the compiler to encode all characters as UTF-8 instead of the native character set.

Representing a Greek letter π requires only two bytes, but with different values than the two low-order bytes in UTF-32: `u8"\xcf\x80"`. An eighth note (♪) requires three or four bytes, again with a different encoding than that used in UTF-32: `u8"\xe2\x99\xaa"` or `u8"\xf0\x9d\x85\xa0"`.

The primary difficulty when dealing with UTF-8 in a program is that the only way to know how many code points are in a string is to scan the entire string. The `size()` member function returns the number of storage units in the string, but each code point requires one to four storage units. On the other hand, UTF-8 has the advantage that you can seek to an arbitrary position in a UTF-8 byte stream and know whether that position is in the middle of a multi-byte character.

UTF-8 is a common encoding for files and network transmissions. It has become the de facto standard for many desktop environments, word processors (including the one I am using to write this book), web pages, and other everyday applications.

Some other environments use UTF-16, which represents a code point using one or two 16-bit integers. The C++ type for a UTF-16 character literal is `char16_t`, and the string type is `std::u16string`. Write such a character literal with the `u` prefix (lowercase letter *u*), for example, `u'\x03c0'`.

Unicode's designers kept the most common code points in the lower 16-bit region (called the *Basic Multilingual Plane,* or BMP). When a code point is outside the BMP, that is, its value exceeds U+FFFF, it requires two storage units in UTF-16 and is called a *surrogate pair*. For example, 𝄞 requires two 16-bit storage units: `u"\xD834\xDD1E"`.

Thus, you have the same problem as UTF-8, namely, that one storage unit does not necessarily represent a single code point, so UTF-16 is less than ideal as an in-memory representation. But the vast majority of code points that most programs deal with fit in a single UTF-16 storage unit, so UTF-16 usually requires half the memory as UTF-32, and in many cases, a `u16string`'s `size()` is the number of code points in the string (although you can't be sure without scanning the string).

Some programmers cope with the difficulty of working with UTF-16 by ignoring surrogate pairs completely. They assume that `size()` does indeed return the number of code points in the string, so their programs work correctly only if all code points are from the BMP. This means you lose access to ancient scripts, specialized alphabets and symbols, and infrequently used ideographs.

UTF-8 has an advantage over UTF-16 and UTF-32 encodings for external representations, because you don't have to deal with endianness. The Unicode standard defines a mechanism for encoding and revealing the endianness of a stream of UTF-16 or UTF-32 text, but that just makes extra work for you.

■ **Note** The position of the most significant byte is called "endianness." A "big-endian" platform is one with the most significant byte first. A "little-endian" platform puts the least significant byte first. The popular Intel x86 platform is little-endian.

Universal Character Names

Unicode makes another official appearance in the C++ standard. You can specify a character using its Unicode code point. Use `\uXXXX` or `\UXXXXXXXX`, replacing `XXXX` or `XXXXXXXX` with the hexadecimal code point. Unlike the `\x` escape, you must use exactly four hexadecimal digits with `\u` or eight with `\U`. These character constructs are called *universal character names*.

Thus, a better way to encode international characters in a string is to use a universal character name. This helps to insulate you against vagaries in the native character set. On the other hand, you have no control over the compiler's actions if it cannot map a Unicode code point to a native character. Therefore, if your native character set is ISO 8859-7 (Greek), the following code should initialize the variable `pi` with the value `'\xf0'`, but if your native character set is ISO 8859-1 (Latin-1), the compiler cannot map it and so might give you a space, a question mark, or something else as, for example:

```
char pi{'\u03c0'};
```

Also note that `\u` and `\U` are not escape sequences (unlike `\x`). You can use them anywhere in a program, not only in a character or string literal. Using a Unicode character name lets you use UTF-8 and UTF-16 strings without knowing the encoding details. Thus, a better way to write the UTF-8 string for Greek lowercase π is: `u8"\u03c0"`, and the compiler will store the encoded bytes, `"\xcf\x80"`.

If you are fortunate, you will be able to avoid universal character names. Instead, your tools will let you edit Unicode characters directly. Instead of dealing with Unicode encoding issues, the editor simply reads and writes universal character names. Thus, the programmer edits WYSIWYG international text, and the source code retains maximum portability. Because universal character names are allowed anywhere, you can use international text in comments too. If you really want to have fun, try using international letters in identifier names. Not all compilers support this feature, although the standard requires it. Thus, you would write a declaration

```
double π{3.14159265358979};
```

and your smart editor would store the following in the source file:

```
double \u03c0{3.14159265358979};
```

and your standard-compliant compiler would accept it and let you use π as an identifier. I don't recommend using extended characters in identifiers unless you know that everyone reading your code is using tools that are aware of universal character names. Otherwise, they make the code much harder to read, understand, and maintain.

Does your compiler support universal character names in strings? ________________ **Does your compiler support universal character names in identifiers?** ________________

Unicode Difficulties

For all the seeming benefits of Unicode, C++ support remains minimal. Although you can write Unicode character literals and string literals, the standard library offers no useful support. Try this exercise: Modify the palindrome program to use `char32_t` instead of `wchar_t`. **What happens?**

____

It doesn't work. There are no I/O stream classes for Unicode. Template specializations for `isalnum` and so on don't exist for `char16_t` or `char32_t`. Although the standard library offers some functions for converting Unicode strings to and from `wstring`, the support ends there.

If you have to work with international characters in any meaningful way, you need a third-party library. The most widely used library is International Components for Unicode (ICU). See the book's web site for a current link.

The next and final topic in Part 3 is to further your understanding of text I/O.

EXPLORATION 56

Text I/O

Input and output have two basic flavors: text and binary. Binary I/O introduces subtleties that are beyond the scope of this book, so all discussion of I/O herein is text-oriented. This Exploration presents a variety of topics related to textual I/O. You've already seen how the input and output operators work with the built-in types as well as with the standard library types, when it makes sense. You've also seen how you can write your own I/O operators for custom types. This Exploration offers some additional details about file modes, reading and writing strings, and converting values to and from strings.

File Modes

Exploration 14 briefly introduced the file stream classes `ifstream` and `ofstream`. The basic behavior is to take a file name and open it. You gain a little more control than that by passing a second argument, which is a file mode. The default mode for an `ifstream` is `std::ios_base::in`, which opens the file for input. The default mode for `ofstream` is `std::ios_base::out | std::ios_base::trunc`. (The | operator combines certain values, such as modes. Exploration 64 will cover this in depth.) The `out` mode opens the file for output. If the file doesn't exist, it is created. The `trunc` mode means to truncate the file, so you always start with an empty file. If you explicitly specify the mode and omit `trunc`, the old contents (if any) remain. Therefore, by default, writing to the output stream overwrites the old contents. If you want to position the stream at the end of the old contents, use the `ate` mode (short for *at-end*), which sets the stream's initial position to the end of the existing file contents. The default is to position the stream at the start of the file.

Another useful mode for output is `app` (short for *append*), which causes every write to append to the file. That is, `app` affects every write, whereas `ate` affects only the starting position. The `app` mode is useful when writing to a log file.

Write a `debug()` function that takes a single string as an argument and writes that string to a file named "debug.txt". Listing 56-1 shows the header that declares the function.

Listing 56-1. Header That Declares a Trivial Debugging Function

```
#ifndef DEBUG_HPP_
#define DEBUG_HPP_

#include <string>

/** @brief Write a debug message to the file @c "debug.txt"
 * @param msg The message to write
 */
void debug(std::string const& msg);

#endif
```

Append every log message to the file, terminating each message with a newline. To ensure that the debugging information is properly recorded, even if the program crashes, open the file anew every time the `debug()` function is called. Listing 56-2 shows my solution.

Listing 56-2. Implementing the Debug Function

```
#include <fstream>
#include <ostream>
#include <stdexcept>

#include <string>
#include "debug.hpp"

void debug(std::string const& str)
{
   std::ofstream stream{"debug.txt", std::ios_base::out | std::ios_base::app};
   if (not stream)
      throw std::runtime_error("cannot open debug.txt");
   stream.exceptions(std::ios_base::failbit);
   stream << str << '\n';
   stream.close();
}
```

String Streams

In addition to file streams, C++ offers string streams. The `<sstream>` header defines `istringstream` and `ostringstream`. A string stream reads from and writes to a `std::string` object. For input, supply the string as an argument to the `istringstream` constructor. For output, you can supply a string object, but the more common usage is to let the stream create and manage the string for you. The stream appends to the string, allowing the string to grow as needed. After you are finished writing to the stream, call the `str()` member function to retrieve the final string.

Suppose you have to read pairs of numbers from a file representing a car's odometer reading and the amount of fuel needed to fill the tank. The program computes the miles per gallon (or liters per kilometer, if you prefer) at each fill-up and overall. The file format is simple: each line has the odometer reading, followed by the fuel amount, on one line, separated by white space.

Write the program. Listing 56-3 demonstrates the miles-per-gallon approach.

Listing 56-3. Computing Miles per Gallon

```
#include <iostream>

int main()
{
   double total_fuel{0.0};
   double total_distance{0.0};
   double prev_odometer{0.0};
   double fuel{}, odometer{};
   while (std::cin >> odometer >> fuel)
   {
      if (fuel != 0)
      {
         double distance{odometer - prev_odometer};
         std::cout << distance / fuel << '\n';
         total_fuel += fuel;
         total_distance += distance;
         prev_odometer = odometer;
      }
   }
   if (total_fuel != 0)
      std::cout << "Net MPG=" << total_distance / total_fuel << '\n';
}
```

Listing 56-4 shows the equivalent program, but computing liters per kilometer. For the remainder of this Exploration, I will use miles per gallon. Readers who don't use this method can consult the files that accompany the book for liters per kilometer.

Listing 56-4. Computing Liters per Kilometer

```
#include <iostream>

int main()
{
   double total_fuel{0.0};
   double total_distance{0.0};
   double prev_odometer{0.0};
   double fuel{}, odometer{};
   while (std::cin >> odometer >> fuel)
   {
      double distance{odometer - prev_odometer};
      if (distance != 0)
      {
         std::cout << fuel / distance << '\n';
         total_fuel += fuel;
         total_distance += distance;
         prev_odometer = odometer;
      }
   }
   if (total_distance != 0)
      std::cout << "Net LPK=" << total_fuel / total_distance << '\n';
}
```

What happens if the user accidentally forgets to record the fuel on one line of the file?

____

The input loop doesn't know or care about lines. It resolutely skips over white space in its quest to fulfill each input request. Thus, it reads the subsequent line's odometer reading as a fuel amount. Naturally, the results will be incorrect.

A better solution would be to read each line as a string and extract two numbers from the string. If the string is not formatted correctly, issue an error message and ignore that line. You read a line of text into a `std::string` by calling the `std::getline` function (declared in `<string>`). This function takes an input stream as the first argument and a `string` object as the second argument. It returns the stream, which means it returns a true value if the read succeeds or a false one if the read fails, so you can use the call to `getline` as a loop condition.

Once you have the string, open an istringstream to read from the string. Use the string stream the same way you would any other input stream. Read two numbers from the string stream. If the string stream does not contain any numbers, ignore that line. If it contains only one number, issue a suitable error message. Listing 56-5 presents the new program.

Listing 56-5. Rewriting the Miles-per-Gallon Program to Parse a String Stream

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
   double prev_odometer{0.0};
   double total_fuel{0.0};
   double total_distance{0.0};
   std::string line{};
   int linenum{0};
   bool error{false};
   while (std::getline(std::cin, line))
   {
      ++linenum;
      std::istringstream input{line};
      double odometer{};
      if (input >> odometer)
      {
         double fuel{};
         if (not (input >> fuel))
         {
            std::cerr << "Missing fuel consumption on line " << linenum << '\n';
            error = true;
         }
         else if (fuel != 0)
         {
            double distance{odometer - prev_odometer};
            std::cout << distance / fuel << '\n';
            total_fuel += fuel;
            total_distance += distance;
            prev_odometer = odometer;
         }
      }
   }
   if (total_fuel != 0)
   {
      std::cout << "Net MPG=" << total_distance / total_fuel;
      if (error)
         std::cout << " (estimated, due to input error)";
      std::cout << '\n';
   }
}
```

Most text file formats allow some form of annotation or commentary. The file format already allows one form of commentary, as a side effect of the program's implementation. **How can you add comments to the input file?**

____

After the program reads the fuel amount from a line, it ignores the rest of the string. You can add comments to any line that contains the proper odometer and fuel data. But that's a sloppy side effect. A better design requires the user to insert an explicit comment marker. Otherwise, the program might misinterpret erroneous input as a valid input, followed by a comment, such as accidentally inserting an extra space, as illustrated in the following:

```
123  21 10.23
```

Let's modify the file format. Any line that begins with a pound sign (#) is a comment. Upon reading a comment character, the program skips the entire line. **Add this feature to the program**. A useful function is an input stream's `unget()` function. After reading a character from the stream, `unget()` returns that character to the stream, causing the subsequent read operation to read that character again. In other words, after reading a line, read a character from the line, and if it is `'#'`, skip the line. Otherwise, call `unget()` and continue as before. Compare your result with mine, as shown in Listing 56-6.

Listing 56-6. Parsing Comments in the Miles-per-Gallon Data File

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
   double total_fuel{0.0};
   double total_distance{0.0};
   double prev_odometer{0.0};
   std::string line{};
   int linenum{0};
   bool error{false};
   while (std::getline(std::cin, line))
   {
      ++linenum;
      std::istringstream input{line};
      char comment{};
      if (input >> comment and comment != '#')
      {
         input.unget();
         double odometer{};
         if (input >> odometer)
         {
           double fuel{};
            if (not (input >> fuel))
            {
               std::cerr << "Missing fuel consumption on line " << linenum << '\n';
               error = true;
            }
            else if (fuel != 0)
            {
               double distance{odometer - prev_odometer};
               std::cout << distance / fuel << '\n';
               total_fuel += fuel;
               total_distance += distance;
               prev_odometer = odometer;
            }
         }
      }
   }
   if (total_fuel != 0)
   {
      std::cout << "Net MPG=" << total_distance / total_fuel;
      if (error)
         std::cout << " (estimated, due to input error)";
      std::cout << '\n';
   }
}
```

More complicated still is allowing the comment marker to appear anywhere on a line. A comment extends from the # character to the end of the line. The comment marker can appear anywhere on a line, but if the line contains any data, it must contain two valid numbers prior to the comment marker. **Enhance the program to allow comment markers anywhere**. Consider using the `find()` member function of `std::string`. It has many forms, one of which takes a character as an argument and returns the zero-based index of the first occurrence of that character in the string. The return type is `std::string::size_type`. If the character is not in the string, `find()` returns the magic constant `std::string::npos`.

Once you find the comment marker, you can delete the comment by calling `erase()` or copy the non-comment portion of the string by calling `substr()`. String member functions work with zero-based indices. Substrings are expressed as a starting position and a count of the number of characters affected. Usually, the count can be omitted to mean the rest of the string. Compare your solution with mine, presented in Listing 56-7.

Listing 56-7. Allowing Comments Anywhere in the Miles-per-Gallon Data File

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
   double total_fuel{0.0};
   double total_distance{0.0};
   double prev_odometer{0.0};
   std::string line{};
   int linenum{0};
   bool error{false};
   while (std::getline(std::cin, line))
   {
      ++linenum;
      std::string::size_type comment{line.find('#')};
      if (comment != std::string::npos)
         line.erase(comment);
      std::istringstream input{line};
      double odometer{};
      if (input >> odometer)
      {
         double fuel{};
         if (not (input >> fuel))
         {
            std::cerr << "Missing fuel consumption on line " << linenum << '\n';
            error = true;
         }
         else if (fuel != 0)
         {
            double distance{odometer - prev_odometer};
            std::cout << distance / fuel << '\n';
            total_fuel += fuel;
            total_distance += distance;
            prev_odometer = odometer;
         }
      }
   }
   if (total_fuel != 0)
   {
      std::cout << "Net MPG=" << total_distance / total_fuel;
      if (error)
         std::cout << " (estimated, due to input error)";
      std::cout << '\n';
   }
}
```

Now that the file format allows explicit comments on each line, you should add some more error-checking to make sure that each line contains only two numbers, and nothing more (after removing comments). One way to check is to read a single character after reading the two numbers. If the read succeeds, the line contains erroneous text. **Add error-checking to detect lines with extra text**. Compare your solution with my solution, shown in Listing 56-8.

Listing 56-8. Adding Error-Checking for Each Line of Input

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
   double total_fuel{0.0};
   double total_distance{0.0};
   double prev_odometer{0.0};
   std::string line{};
   int linenum{0};
   bool error{false};
   while (std::getline(std::cin, line))
   {
      ++linenum;
      std::string::size_type comment{line.find('#')};
      if (comment != std::string::npos)
         line.erase(comment);
      std::istringstream input{line};
      double odometer{};
      if (input >> odometer)
      {
         double fuel{};
         char check{};
         if (not (input >> fuel))
         {
            std::cerr << "Missing fuel consumption on line " << linenum << '\n';
            error = true;
         }
         else if (input >> check)
         {
            std::cerr << "Extra text on line " << linenum << '\n';
            error = true;
         }
         else if (fuel != 0)
         {
            double distance{odometer - prev_odometer};
            std::cout << distance / fuel << '\n';
            total_fuel += fuel;
            total_distance += distance;
            prev_odometer = odometer;
         }
      }
   }
   if (total_fuel != 0)
   {
      std::cout << "Net MPG=" << total_distance / total_fuel;
      if (error)
         std::cout << " (estimated, due to input error)";
      std::cout << '\n';
   }
}
```

Text Conversion

Let me put on my clairvoyance cap for a moment . . . I can see that you have many unanswered questions about C++; and one of those questions is, “How can I convert a number to a string easily, and vice versa?”

The standard library offers some simple functions: `std::to_string()` takes an integer and returns a string representation. To convert a string to an integer, choose from several functions, depending on the desired return type: `std::stoi()` returns an `int`, and `std::stod()` returns `double`.

But these functions offer little flexibility. You know that I/O streams offer lots of flexibility and control over formatting. Surely, you say, you can create functions that are just as easy to use with suitable defaults, but also offer some flexibility in formatting (such as floating-point precision, fill characters, etc.).

Now that you know how to use string streams, the way forward is clear: use an `istringstream` to read a number from a string, or use an `ostringstream` to write a number to a string. The only task is to wrap up the functionality in an appropriate function. Even better is to use a template. After all, reading or writing an `int` is essentially the same as reading or writing a `long`, etc.

Listing 56-9 shows the `from_string` function template, which has a single template parameter, `T`—the type of object to convert. The function returns type `T` and takes a single function argument: a string to convert.

Listing 56-9. The `from_string` Function Extracts a Value from a String

```
#include <istream> // for the >> operator
#include <sstream> // for istringstream
#include <string>  // for string
#include "conversion_error.hpp"

template<class T>
T from_string(std::string const& str)
{
  std::istringstream in{str};
  T result{};
  if (in >> result)
    return result;
  else
    throw conversion_error{str};
}
```

The `conversion_error` class is a custom exception class. The details are not relevant to this discussion, but inquisitive readers can satisfy their curiosity with the files that accompany this book. `T` can be any type that permits reading from an input stream with the `>>` operator, including any custom operators and types that you write.

Now what about the advertised flexibility? Let's add some. As written, the `from_string` function does not check for text that follows the value. Plus, it skips over leading white space. **Modify the function to take a bool argument: `skipws`**. If true, `from_string` skips leading white space and allows trailing white space. If false, it does not skip leading white space, and it does not permit trailing white space. In both cases, it throws `conversion_error`, if invalid text follows the value. Compare your solution with mine in Listing 56-10.

Listing 56-10. Enhancing the `from_string` Function

```
#include <ios>     // for std::noskipws
#include <istream> // for the >> operator
#include <sstream> // for istringstream
#include <string>  // for string
#include "conversion_error.hpp"

template<class T>
T from_string(std::string const& str, bool skipws = true)
{
  std::istringstream in{str};
  if (not skipws)
    in >> std::noskipws;
  T result{};
  char extra;
  if (not (in >> result))
    throw conversion_error{str};
  else if (in >> extra)
    throw conversion_error{str};
  else
    return result;
}
```

I threw in a new language feature. The function parameter skipws is followed by = true, which looks like an assignment or initialization. It lets you call from_string with one argument, just as before, using true as the second argument. This is how file streams specify a default file mode, in case you were wondering. If you decide to declare default argument values, you must supply them starting with the right-most argument in the argument list. I don't use default arguments often, and in Exploration 69, you will learn about some subtleties related to default arguments and overloading. For now, use them when they help, but use them sparingly.

Your turn to write a function from scratch. **Write the to_string function template**, which takes a single template argument and declares the to_string function to take a single function argument of that type. The function converts its argument to a string by writing it to a string stream and returns the resulting string. Compare your solution with mine, presented in Listing 56-11.

Listing 56-11. The to_string Function Converts a Value to a String

```
#include <ostream> // for the << operator
#include <sstream> // for ostringstream
#include <string>  // for string

template<class T>
std::string to_string(T const& obj)
{
  std::ostringstream out{};
  out << obj;
  return out.str();
}
```

Can you see any particular drawback to these functions? ________________ If so, what?

____

No doubt, you can see many problems, but in particular, the one I want to point out is that they don't work with wide characters. It would be a shame to waste all that effort you spent in understanding wide characters, so let's add another template parameter for the character type. Remember that the std::string class template has three template parameters: the character type, the character traits, and an allocator object to manage any heap memory that the string might use. You don't have to know any of the details of these three types; you need only pass them to the basic_string class template. The basic_ostringstream class template takes the first two template arguments.

Your first attempt at implementing to_string may look a little bit like Listing 56-12.

Listing 56-12. Rewriting to_string As a Template Function

```
#include <ostream> // for the << operator
#include <sstream> // for ostringstream
#include <string>  // for basic_string

template<class T, class Char, class Traits, class Allocator>
std::basic_string<Char, Traits, Allocator> to_string(T const& obj)
{
  std::basic_ostringstream<Char, Traits> out{};
  out << obj;
  return out.str();
}
```

This implementation works. It's correct, but it's clumsy. Try it. **Try to write a simple test program that converts an integer to a narrow string and the same integer to a wide string**. Don't be discouraged if you can't figure out how to do it. This exercise is a demonstration of how templates in the standard library can lead you astray if you aren't careful. Take a look at my solution in Listing 56-13.

Listing 56-13. Demonstrating the Use of `to_string`

```
#include <iostream>
#include "to_string.hpp"
#include "from_string.hpp"

int main()
{
    std::string str{
      to_string<int, char, std::char_traits<char>, std::allocator<char>>(42)
    };
    int value{from_string<int>(str)};
}
```

Do you see what I mean? How were you supposed to know what to provide as the third and fourth template arguments? Don't worry, we can find a better solution.

One alternative approach is not to return the string, but to take it as an output function argument. The compiler could then use argument-type deduction, and you wouldn't have to specify all those template arguments. **Write a version of `to_string` that takes the same template parameters** but also two function arguments: the value to convert and the destination string. **Write a demonstration program** to show how much simpler this function is to use. Listing 56-14 shows my solution.

Listing 56-14. Passing the Destination String As an Argument to `to_string`

```
#include <ostream> // for the << operator
#include <sstream> // for ostringstream
#include <string>  // for string
#include "from_string.hpp"

template<class T, class Char, class Traits, class Allocator>
void to_string(T const& obj, std::basic_string<Char, Traits, Allocator>& result)
{
  std::basic_ostringstream<Char, Traits> out{};
  out << obj;
  result = out.str();
}

int main()
{
    std::string str{};
    to_string(42, str);
    int value(from_string<int>(str));
}
```

On the other hand, if you want to use the string in an expression, you still have to declare a temporary variable just to hold the string.

Another way to approach this problem is to specify `std::string` or `std::wstring` as the sole template argument. The compiler can deduce the type of the object you want to convert. The `basic_string` template has member typedefs for its template parameters, so you can use those to discover the traits and allocator types. The `to_string` function returns the string type and takes an argument of the object type. Both types have to be template parameters. **Which parameter should be first?** Listing 56-15 shows the latest incarnation of `to_string`, which now takes two template parameters: the string type and the object type.

Listing 56-15. Improving the Calling Interface of `to_string`

```
#include <ostream> // for the << operator
#include <sstream> // for ostringstream

template<class String, class T>
String to_string(T const& obj)
{
  std::basic_ostringstream<typename String::value_type,
                           typename String::traits_type> out{};
  out << obj;
  return out.str();
}
```

Remember `typename` from Exploration 53? The compiler doesn't know that `String::value_type` names a type. A specialization of `basic_ostringstream` could declare it to be anything. The `typename` keyword tells the compiler that you know the name is for a type. Calling this form of `to_string` is straightforward.

```
to_string<std::string>(42);
```

This form seems to strike the best balance between flexibility and ease-of-use. But can we add some more formatting flexibility? Should we add width and fill characters? Field adjustment? Hexadecimal or octal? What if `to_string` takes `std::ios_base::fmtflags` as an argument, and the caller can specify any formatting flags? What should be the default? Listing 56-16 shows what happens when the author goes overboard.

Listing 56-16. Making `to_string` Too Complicated

```
#include <ios>
#include <ostream> // for the << operator
#include <sstream> // for ostringstream

template<class String, class T>
String to_string(T const& obj,
  std::ios_base::fmtflags flags = std::ios_base::fmtflags{},
  int width = 0,
  char fill = ' ')
{
  std::basic_ostringstream<typename String::value_type,
                           typename String::traits_type> out{};
  out.flags(flags);
  out.width(width);
  out.fill(fill);
  out << obj;
  return out.str();
}
```

Listing 56-17 shows some examples of calling this form of `to_string`.

Listing 56-17. Calling `to_string`

```
#include <iostream>
#include "to_string.hpp"

int main()
{
  std::cout << to_string<std::string>(42, std::ios_base::hex) << '\n';
  std::cout << to_string<std::string>(42.0, std::ios_base::scientific, 10) << '\n';
  std::cout << to_string<std::string>(true, std::ios_base::boolalpha) << '\n';
}
```

You should see the following as output:

```
2a

4.200000e+01
true
```

That concludes Part 3. Time for a project.

EXPLORATION 57

Project 3: Currency Type

It's time for another project. You're going to continue building on the `fixed` type from Project 2 and incorporate what you've learned about locales and I/O. Your task this time is to write a `currency` type. The value is stored as a fixed-point value. I/O is formatted using the `get_money` and `put_money` manipulators.

Make sure you can add two `currency` amounts to get a `currency` value, subtract two `currency` amounts to get `currency`, multiply and divide `currency` by an integer or `rational` value to get a `currency` result, and divide two `currency` values to get a `rational` result.

As with any project, start small and add functionality as you go. For example, start with the basic data representation, then add I/O operators. Add arithmetic operators one at a time. Write each test function before you implement the feature.

EXPLORATION 58

Pointers

Few topics cause more confusion, especially for programmers new to C++, than pointers. Necessary, powerful, and versatile, pointers can also be dangerous and the underlying cause of so many bugs that they are both bane and blessing. Pointers are hard at work behind many of the standard library's features, and any serious application or library inevitably uses pointers in some fashion. When used with care and caution, pointers will become an indispensable tool in your C++ programmer's toolkit.

A Programming Problem

Before diving into syntax and semantics, consider the following problem. Real-life C++ projects typically contain multiple source files, and each source file includes multiple headers. While you are working, you will compile and recompile the project many times. Each time, it's preferable to recompile only those files that have changed or include a header file that has changed. Different development environments have different tools to decide which files to recompile. An IDE typically makes these decisions itself; in other environments, a separate tool, such as `make`, `jam`, or `scons`, examines the files in your project and decides which ones to recompile.

The problem to tackle in this and following Explorations is to write a simple tool that decides which files to compile and pretends to compile them. (Actually invoking an external program is beyond the scope of this book, so you won't learn how to write an entire build tool.)

The essential idea is simple: to make an executable program, you must compile source files into object files and link the object files to form the program. The executable program is said to *depend on* the object files, which in turn, depend on the source files. Other terminology has the program as the *target*, with the object files as its *dependencies*. An object file, in turn, can be a target, with a source file and the header files that it includes as dependencies.

As you know, to compile a single source file into a single object file, the compiler may be required to read many additional header files. Each of these header files is a dependency of the object file. Thus, one header file can be a dependency of many object files. In more technical terms, targets and dependencies form a directed acyclic graph (DAG), which I will call the *dependency graph*.

Note A cyclic graph, such that A depends on B, and B depends on A, is a bad idea in the real world and often indicates a faulty or poorly considered design. For the sake of simplicity, I will ignore this error condition in this and subsequent Explorations.

Anyone who's been around large projects knows that dependency graphs can become extremely complex. Some header files may be generated by other programs, so the header files are targets, with the generating programs as dependencies, and the generating programs are targets with their own dependencies.

IDEs and programs, such as make, analyze the dependency graph and determine which targets must be built first to ensure that every target's dependencies are fulfilled. Thus, if A depends on B, and B depends on C, make must build C first (if it is a target), then B, and finally A. The key algorithm that make employs to find the correct order in which to build targets is a *topological sort*.

Topological sorts are not included in the typical algorithms coursework of many computer science majors. Nor does the algorithm appear in many textbooks. However, any comprehensive algorithms book includes topological sort.

■ **Note** A good text on topological sort is *Introduction to Algorithms*, 3rd ed., by T. H. Cormen, C. E. Leiserson, and R. L. Rivest (MIT Press, 2009). My solution implements exercise 22.4-5.

The C++ standard library does not include a topological sort algorithm, because it is not a sequential algorithm. It operates on a graph, and the C++ library has no standard graph classes. (Boost has a graph library that includes topological sort, but to ensure everyone can use this Exploration, we will write our own topological sort function.)

We'll begin this Exploration by writing a pseudo-make program—that is, a program that reads a *makefile*: a file that describes a set of targets and their dependencies, performs a topological sort to find the order for building targets, and prints the targets in proper build order. In order to simplify the program somewhat, restrict the input to a text file that declares dependencies as pairs of strings, one pair on a line of text. The first string is the name of a target, and the second string is the name of a dependency. If a target has multiple dependencies, the input file must list the target on multiple lines, one per dependency. A target can be a dependency of another target. The order of lines within the input file is not important. The goal is to write a program that will print the targets in order, so that a make-like program can build the first target first, and proceed in order, such that all targets are built before they are needed as dependencies.

To help clarify terminology, I use the term *artifact* for a string that can be a target, a dependency, or both. If you already know an algorithm for topological sort, go ahead and implement the program now. Otherwise, keep reading to see one implementation of topological_sort. To represent the dependency graph, use a map of sets. The map key is a dependency, and the value is the set of targets that list the key as a dependency. This seems inside out from the way you may usually think about organizing targets and dependencies, but as you can see in Listing 58-1, it makes the topological sort quite easy to implement. Because the topological_sort function is reusable, it is a template function and works with *nodes* instead of artifacts, targets, and dependencies.

Listing 58-1. Topological Sort of a Directed Acyclic Graph

```
#ifndef TOPOLOGICAL_SORT_HPP_
#define TOPOLOGICAL_SORT_HPP_

#include <deque>
#include <stdexcept>

/// Helper function for topological_sort().
/// Finds all the nodes in the graph with no incoming edges,
/// that is, with empty values. Removes each one from the graph
/// and adds it to the set @p nodes.
/// @param[in,out] graph A map of node/set pairs
/// @param[in,out] nodes A queue of nodes with no incoming edges
template<class Graph, class Nodes>
void topsort_clean_graph(Graph& graph, Nodes& nodes)
```

```
{
  for (auto iter(graph.begin()), end(graph.end()); iter != end;)
  {
    if (iter->second.empty())
    {
      nodes.push_back(iter->first);
      graph.erase(iter++);  // advance iterator before erase invalidates it
    }
    else
      ++iter;
  }
}

/// Topological sort of a directed acyclic graph.
/// A graph is a map keyed by nodes, with sets of nodes as values.
/// Edges run from values to keys. The sorted list of nodes
/// is copied to an output iterator in reverse order.
/// @param graph The graph
/// @param sorted The output iterator
/// @throws std::runtime_error if the graph contains a cycle
/// @pre Graph::key_type == Graph::mapped_type::key_type
template<class Graph, class OutIterator>
void topological_sort(Graph graph, OutIterator sorted)
{
  std::deque<typename Graph::key_type> nodes{};
  // Start with the set of nodes with no incoming edges.
  topsort_clean_graph(graph, nodes);

  while (not nodes.empty())
  {
    // Grab the first node to process, output it to sorted,
    // and remove it from the graph.
    typename Graph::key_type n{nodes.front()};
    nodes.pop_front();
    *sorted = n;
    ++sorted;

    // Erase n from the graph
    for (auto& node : graph)
    {
      node.second.erase(n);
    }
    // After removing n, find any nodes that no longer
    // have any incoming edges.
    topsort_clean_graph(graph, nodes);
  }
  if (not graph.empty())
    throw std::invalid_argument("Dependency graph contains cycles");
}

#endif // TOPOLOGICAL_SORT_HPP_
```

Now that you have the `topological_sort` function, **implement the pseudo-`make` program** to read and parse the input, build the dependency graph, call `topological_sort`, and print the sorted result. Keep things simple and treat artifacts (targets and dependencies) as strings. Thus, the dependency graph is a map with `std::string` as the key type and `std::set<std::string>` as the value type. Compare your solution with Listing 58-2.

Listing 58-2. First Draft of the Pseudo-make Program

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <sstream>
#include <stdexcept>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

#include "topological_sort.hpp"

typedef std::string artifact; ///< A target, dependency, or both

class dependency_graph
{
public:
  typedef std::unordered_map<artifact, std::unordered_set<artifact>> graph_type;

  void store_dependency(artifact const& target, artifact const& dependency)
  {
    graph_[dependency].insert(target);
    graph_[target]; // ensures that target is in the graph
  }

  graph_type const& graph() const { return graph_; }

  template<class OutIter>
  void sort(OutIter sorted)
  const
  {
    topological_sort(graph_, sorted);
  }

private:
  graph_type graph_;
};

int main()
{
  dependency_graph graph{};

  std::string line{};
  while (std::getline(std::cin, line))
  {
    std::string target{}, dependency{};
    std::istringstream stream{line};
    if (stream >> target >> dependency)
      graph.store_dependency(target, dependency);
    else if (not target.empty())
      // Input line has a target with no dependency,
      // so report an error.
      std::cerr << "malformed input: target, " << target <<
                   ", must be followed by a dependency name\n";
    // else ignore blank lines
  }

  try {
    // Get the artifacts. The sort stores them in reverse order.
    std::vector<artifact> sorted{};
    graph.sort(std::back_inserter(sorted));
    // Then print them in the correct order.
    std::copy(sorted.rbegin(), sorted.rend(),
              std::ostream_iterator<artifact>(std::cout, "\n"));
  } catch (std::runtime_error const& ex) {
    std::cerr << ex.what() << '\n';
    return EXIT_FAILURE;
  }
}
```

So what do DAGs and topological sorts have to do with the topic of this Exploration? I thought you'd never ask. Let's construct a slightly more complicated problem by making it a little more realistic.

A real make program has to keep track of more information about an artifact, especially the time when it was last modified. A target also has a list of actions to perform, if any dependency is newer than the target. Thus, a class makes more sense than a string for representing an artifact. You can add to the class whatever functionality you need for your make program.

Standard C++ does not have any functions for querying a file's modification time. For now, we'll just sidestep the issue and make up a time for every artifact. The important task at hand is to associate additional information with an artifact. The <chrono> header declares a time type that goes by the rather unwieldy qualified name of

```
std::chrono::system_clock::time_point
```

Ignoring actions, you might define the `artifact` type as shown in Listing 58-3.

Listing 58-3. New Definition of an Artifact

```
#ifndef ARTIFACT_HPP_
#define ARTIFACT_HPP_

#include <chrono>
#include <string>

class artifact
{
public:
  typedef  std::chrono::system_clock clock;
  artifact() : name_{}, mod_time_{clock::time_point{}} {}
  artifact(std::string const& name)
  : name_{name}, mod_time_{get_mod_time()}
  {}

  std::string const& name()     const { return name_; }
  clock::time_point  mod_time() const { return mod_time_; }

  /// Build a target.
  /// After completing the actions (not yet implemented),
  /// update the modification time.
  void build();

  /// Look up the modification time of the artifact.
  /// Return time_point{} if the artifact does not
  /// exist (and therefore must be built) or if the time cannot
  /// be obtained for any other reason.
  /// Also see boost::file_system::last_write_time.
  clock::time_point get_mod_time()
  {
    // Real programs should get this information from the
    // operating system. This program returns the current time.
    return clock::now();
  }
private:
  std::string name_;
  clock::time_point mod_time_;
};

#endif // ARTIFACT_HPP_
```

Now we run into a problem. In the first draft of this program, what made two strings refer to the same artifact is that the strings had the same content. The target named "program" is the same artifact as the dependency named "program," because they are spelled the same. That scheme falls down now that an artifact is more than just a string. When you build a target and update its modification time, you want all uses of that artifact to be updated. Somehow, every use of an artifact name must be associated with a single artifact object for that name.

Got any ideas? It can be done with your current understanding of C++, but you may have to stop and think about it.

Need a hint? How about storing all artifacts in one big vector? Then make a dependency graph that contains indices into the vector, instead of artifact names. **Try it**. Rewrite the program in Listing 58-2 to use the new `artifact.hpp` header from Listing 58-3. When an artifact name is read from the input file, look up that name in a vector of all artifacts. If the artifact is new, add it to the end. Store vector indices in the dependency graph. Print the final list by looking up the numbers in the vector. Compare your solution with Listing 58-4.

Listing 58-4. Second Draft, Adding Modification Times to Artifacts

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <sstream>
#include <stdexcept>
```

```
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

#include "artifact.hpp"
#include "topological_sort.hpp"

typedef std::size_t artifact_index;

class dependency_graph
{
public:
  typedef std::unordered_map<artifact_index, std::unordered_set<artifact_index>> graph_type;

  void store_dependency(artifact_index target, artifact_index dependency)
  {
    graph_[dependency].insert(target);
    graph_[target]; // ensures that target is in the graph
  }

  graph_type const& graph() const { return graph_; }

  template<class OutIter>
  void sort(OutIter sorted)
  const
  {
    topological_sort(graph_, sorted);
  }

private:
  graph_type graph_;
};

std::vector<artifact> artifacts{};
artifact_index lookup_artifact(std::string const& name)
{
  auto iter( std::find_if(artifacts.begin(), artifacts.end(),
    [&name](artifact const& a) { return a.name() == name; })
  );
  if (iter != artifacts.end())
    return iter - artifacts.begin();
  // Artifact not found, so add it to the end.
  artifacts.push_back(artifact{name});
  return artifacts.size() - 1;
}

int main()
{
  dependency_graph graph{};
  std::string line{};
  while (std::getline(std::cin, line))
  {
    std::string target_name{}, dependency_name{};
    std::istringstream stream{line};
    if (stream >> target_name >> dependency_name)
    {
      artifact_index target{lookup_artifact(target_name)};
      artifact_index dependency{lookup_artifact(dependency_name)};
      graph.store_dependency(target, dependency);
    }
    else if (not target_name.empty())
      // Input line has a target with no dependency,
      // so report an error.
      std::cerr << "malformed input: target, " << target_name <<
                   ", must be followed by a dependency name\n";
    // else ignore blank lines
  }

  try {
    // Get the sorted artifact indices in reverse order.
    std::vector<artifact_index> sorted{};
    graph.sort(std::back_inserter(sorted));
    // Then print the artifacts in the correct order.
    for (auto it(sorted.rbegin()), end(sorted.rend());
         it != end;
         ++it)
    {
      std::cout << artifacts.at(*it).name() << '\n';
    }
  } catch (std::runtime_error const& ex) {
    std::cerr << ex.what() << '\n';
    return EXIT_FAILURE;
  }
}
```

■ **Note** If the performance of linear lookups concerns you, congratulations for sharp thinking. Not to worry, however, because the program will continue to grow and evolve throughout this Exploration, and we will eliminate the performance issue before we finish.

Well, that works, but it's ugly. Looking up indices is sloppy programming. Much better would be to store references to the `artifact` objects directly in the graph. Ah, there's the rub. You can't store a reference in a standard container. Containers are for storing objects—real objects. The container must be able to copy and assign the elements in the container, but it can't do that with references. Copying a reference actually copies the object to which it refers. A reference is not a first-class entity that a program can manipulate.

Wouldn't it be nice if C++ had a language feature that acts like a reference but lets you copy and assign the reference itself (not the referred-to object)? Let's pretend we are inventing the C++ language, and we have to add this language feature.

The Solution

Let's devise a new language feature to solve this programming problem. This new feature is similar to references but permits use with standard containers. Let's call this feature a *flex-ref*, short for "flexible reference."

If `a` and `b` are both flex-refs that refer to type `int`, the statement

```
a = b;
```

means the value of `a` is changed so that `a` now refers to the same `int` object to which b refers. Passing `a` as an argument to a function passes the value of `a`, so if the function assigns a new value to `a`, that change is local to the function (just as with any other function argument). Using a suitable operator, however, the function can obtain the `int` object to which `a` refers, and read or modify that `int`.

You need a way to obtain the referred-to value, so we have to invent a new operator. Look at iterators for inspiration: given an iterator, the unary * operator returns the item to which the iterator refers. Let's use the same operator for flex-refs. Thus, the following prints the `int` value to which `a` refers:

```
std::cout << *a;
```

In the spirit of the * operator, declare a flex-ref by using * in the same manner that you use `&` for references.

```
int *a, *b;
```

Use the same syntax when declaring a container. For example, declare a vector of flex-refs that refer to type `int`.

```
std::vector<int*> vec;
vec.push_back(a);
b = vec.front();
```

All that's left is to provide a way to make a flex-ref refer to an object. For that, let's turn to ordinary references for inspiration and use the `&` operator. Supposing that `c` is of type `int`, the following makes `a` refer to `c`:

```
a = &c;
```

As you've guessed by now, flex-refs are pointers. The variables `a` and `b` are called "pointers to `int`." A pointer is an honest-to-goodness lvalue. It occupies memory. The values that are stored in that memory are addresses of other lvalues. You can freely change the value stored in that memory, which has the effect of making the pointer refer to a different object.

A pointer can point to a `const` object, or it can be a `const` pointer, or both. The following shows pointer to `const int`:

```
int const* p;
p = &c;
```

Define a const pointer—that is, a pointer that is itself const and therefore cannot be the target of an assignment, but the dereferenced object can be the target.

```
int * const q{&c};
*q = 42;
```

Like any const object, you must supply an initializer, and you cannot modify the pointer. However, you can modify the object to which the pointer points.

You can define a reference to a pointer, just as you can define a reference to anything (except another reference).

```
int const*&r{p};
```

Read this declaration the way you would read any other declaration: start by finding the declarator, r. Then read the declaration from the inside, working your way outward. To the left, see &, telling you that r is a reference. To the right is the initializer, {p}; r is a reference to p (r is another name for the object p). Continuing to the left, you see *, so r is a reference to a pointer. Then you see const, so you know r is a reference to a pointer to a const something. Finally, int tells you that r is a reference to a pointer to const int. Thus, the initializer is valid, because its type is pointer to int.

What about the other way around? Can you define a pointer to a reference? The short answer is that you can't. A pointer to a reference makes as little sense as a reference to a reference. References and pointers must refer or point to a real object. When you use the & operator on a reference, you get the address of the referenced object.

You can define a pointer to a pointer. Or a pointer to a pointer to a pointer to a pointer. . . Just keep track of the exact type of your pointer. The compiler ensures that you assign only expressions of the correct type, as shown in the following:

```
int x;
int *y;
int **z;
y = &x;
z = &y;
```

Try z = &x and y = z. What happens?

____

Because x has type int, &x has type int*; y has type int*, too, so you can assign &x to y but not to z, which has type int**. The types must match, so you can't assign z to y either.

It took me long enough to get to the point, but now you can see how pointers help solve the problem of writing the dependency graph. Before we dive into the code, however, let's take a moment to clarify some terminology.

Addresses vs. Pointers

Programmers are sticklers for details. The compilers and other tools we use daily force us to be. So let's be absolutely clear about addresses and pointers.

An *address* is a memory location. In C++ parlance, it is an rvalue, so you cannot modify or assign to an address. When a program takes the address of an object (with the & operator), the result is a constant for the lifetime of that object. Like every other rvalue, an address in C++ has a type, which must be a pointer type.

A *pointer type* is more properly called an *address type,* because the range of values represented by the type are addresses. Nonetheless, the term *pointer type* is more common, because a pointer object has a pointer type.

A pointer type can denote multiple levels of indirection—it can denote a pointer to a pointer, or a pointer to a pointer to a pointer, etc. You must declare each level of pointer indirection with an asterisk. In other words, int* is the type "pointer to int" and int** is "pointer to pointer to int."

A *pointer* is an lvalue that has a pointer type. A pointer object, like any object, has a location in memory in which the program can store a value. The value must have a type that is compatible with the pointer's type; the value must be an address of the correct type.

Dependency Graphs

Now let's get back to the dependency graph. The graph can store pointers to artifacts. Each external file corresponds to a single `artifact` object in the program. That artifact can have many nodes in the graph pointing to it. If you update that artifact, all nodes that point to the artifact see the update. Thus, when a build rule updates an artifact, the file modification time may change. All nodes for that artifact in the graph immediately see the new time, because they all point to a single object.

All that's left to figure out is where these artifacts reside. For the sake of simplicity, I recommend a `map`, keyed by artifact name. The mapped values are `artifact` objects (not pointers). Take the address of an artifact in the map to obtain pointers to store in the graph. **Go ahead; don't wait for me**. Using the `topological_sort.hpp` and `artifact.hpp` headers, **rewrite Listing 58-4** to store `artifact` objects in a map and `artifact` pointers in the graph. Compare your solution with Listing 58-5.

Listing 58-5. Storing Pointers in the Dependency Graph

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <map>
#include <sstream>
#include <stdexcept>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

#include "artifact.hpp"
#include "topological_sort.hpp"

class dependency_graph
{
public:
  typedef std::unordered_map<artifact*, std::unordered_set<artifact*>> graph_type;

  void store_dependency(artifact* target, artifact* dependency)
  {
    graph_[dependency].insert(target);
    graph_[target]; // ensures that target is in the graph
  }

  graph_type const& graph() const { return graph_; }
  template<class OutIter>
  void sort(OutIter sorted)
  const
  {
    topological_sort(graph_, sorted);
  }

private:
  graph_type graph_;
};

std::map<std::string, artifact> artifacts{};

artifact* lookup_artifact(std::string const& name)
{
  auto a( artifacts.find(name) );
  if (a == artifacts.end())
    artifacts.emplace(name, artifact{name});
  return &artifacts[name];
}

int main()
{
  dependency_graph graph{};

  std::string line{};
  while (std::getline(std::cin, line))
  {
    std::string target_name{}, dependency_name{};
    std::istringstream stream{line};
    if (stream >> target_name >> dependency_name)
    {
      artifact* target{lookup_artifact(target_name)};
      artifact* dependency{lookup_artifact(dependency_name)};
      graph.store_dependency(target, dependency);
    }
    else if (not target_name.empty())
      // Input line has a target with no dependency, so report an error.
      std::cerr << "malformed input: target, " << target_name <<
                   ", must be followed by a dependency name\n";
    // else ignore blank lines
  }

  try {
    // Get the sorted artifacts in reverse order.
    std::vector<artifact*> sorted{};
    graph.sort(std::back_inserter(sorted));
    // Then print the artifacts in the correct order.
    for (auto it(sorted.rbegin()), end(sorted.rend());
         it != end;
         ++it)
    {
      std::cout << (*it)->name() << '\n';
    }
  } catch (std::runtime_error const& ex) {
    std::cerr << ex.what() << '\n';
    return EXIT_FAILURE;
  }
}
```

Dereferencing the iterator yields a pointer. To call the `name()` member function from a pointer, use the arrow (`->`) operator. The trick is that `->` has higher precedence than `*`, so you must put `*it` in parentheses in order to dereference the iterator first. Overall, the program requires minimal changes, and the changes are mostly simplifications. As the program grows more complicated (as real programs inevitably do), the simplicity and elegance of pointers become more and more evident.

Standard containers are extremely helpful, but sometimes a program needs greater control over its objects. It must create and destroy objects on its own schedule, not when functions start and end or when control enters or leaves a block. The next Exploration tackles this problem by introducing dynamic memory.

EXPLORATION 59

Dynamic Memory

Declaring pointers is all well and good, but real programs have to do more. The next step is to learn how to create new objects on the fly, at runtime. Your program takes full control over the lifetime of these objects, destroying the objects only when the program is done using them. This Exploration details how to allocate and free memory dynamically. It also continues to develop artifact and related classes from Exploration 58.

I want to warn you, however, not to run off immediately and start using dynamic memory in your programs. We still have several more Explorations to go, each one building on its predecessors. You need the full picture, which will include safer ways to manage pointers and dynamic memory.

Allocating Memory

A new expression allocates memory for a new object, calls a constructor to initialize that object, and returns the address of the newly allocated and constructed object. The syntax is the new keyword, followed by the base type, followed by an optional initializer.

```
int *pointer{new int{42}};
std::cout << *pointer << '\n'; // prints 42
*pointer = 10;                 // changes the int to 10
```

The new expression returns the address of an lvalue. The type of the lvalue is the type you provide after the new keyword. The normal rules for initialization apply. If the initializer is a set of empty curly braces, the newly allocated object is zero-initialized (see Exploration 32), meaning all members are initialized to known, zero values. If you omit the curly braces entirely, the new object is default-initialized, which means objects of built-in type are left uninitialized. This is usually a bad thing, so I recommend providing an initializer with all new expressions, such as shown in the following:

```
int *good{new int{}}; // *good == 0
int *bad{new int};    // undefined behavior if you read *bad
```

As with any other initializer, if you are initializing a class-type object, and the constructor takes multiple arguments, separate the arguments with commas.

```
rational* piptr{new rational{355, 113}};
```

If the C++ environment runs out of memory and cannot fulfill a new request, it throws the std::bad_alloc exception (declared in <new>).

Once allocated and initialized, a dynamically allocated object lives until you get rid of it with a delete expression (as covered in the next section).

Freeing Memory

When your program no longer requires an object that it had allocated with a new expression, it must free that object with a delete expression. The syntax is the delete keyword, followed by a pointer to the object you want to delete.

```
int *tmp{new int{42}};
// use *tmp
delete tmp;
```

After you delete an object, all pointers to it become invalid. It is your responsibility to ensure that you never dereference, copy, or otherwise use these pointers. The only thing you can do with a pointer variable after you delete its object is to assign a new value to the variable.

Dynamic objects are more difficult to work with than other objects, because they impose a greater burden on you, the programmer, to manage their lifetimes. Almost any mistake results in undefined behavior. A few of the most common mistakes are

- Using a pointer after a delete
- Invoking delete more than once on the same object
- Failing to delete an object before the program terminates

Most programmers are familiar with segmentation faults, access violations, and the like. These are the most benign of the actual behavior you might encounter if you fail to follow these rules.

It sure would be nice to be able to test whether a pointer is valid before dereferencing it. Too bad you can't. The only way to ensure that your program never dereferences an invalid pointer or tries to otherwise misuse a pointer is to be very, very careful when you write programs that use pointers and dynamic memory. That's why I waited until late in the book to introduce these topics.

Fortunately, C++ offers some tools to help you: one is a special pointer value that you can assign to any pointer variable to represent a "pointer to nothing." You cannot dereference a pointer to nothing, but you can copy and assign these pointers, and most important, compare a pointer variable with the "pointer to nothing" value. In other words, when your program deletes an object, it should assign a "pointer to nothing" value to the pointer variable. By ensuring that every pointer variable stores a valid pointer or a "pointer to nothing," you *can* safely test whether a pointer is valid before dereferencing it.

Pointer to Nothing

A "pointer to nothing" is called a *null pointer,* and you write a null pointer with the nullptr keyword. Use nullptr to initialize a pointer variable, as the source of an assignment or as a function argument, as shown in the following:

```
int* ptr{nullptr};
ptr = nullptr;
```

If you initialize a pointer with empty curly braces, you also get a null pointer. That is, the following declarations are equivalent:

```
int* ptr1{};
int* ptr2{nullptr};
```

I prefer being explicit, so I always use nullptr.

OLD-FASHIONED NULL POINTERS

In C++ 03, a null pointer was written with an integer literal 0. Before that, C programmers used NULL for null pointers. Unfortunately, a few errant C and C++ programmers also used NULL to mean zero in non-pointer expressions. Some C++ programmers continued to use NULL instead of 0. As you might imagine, this led to all kinds of problems, leading to the invention of the nullptr keyword.

All new code should use nullptr. The addition of the nullptr keyword is such an important leap forward in clarity that I go out of my way to fix old code to use nullptr. Ordinarily, I don't take existing, working code and go in to add C++ 11 features, but I make an exception for nullptr.

The bits actually stored in a pointer variable are implementation-defined. In particular, you cannot make any assumption about the bits used to represent nullptr. But whatever those bits are, C++ will use the correct address for your environment when you zero-initialize a pointer or use nullptr.

You have one other guarantee with null pointers: whatever value the C++ environment uses for them, no real object will ever have that value as its address. Thus, you can assign a null pointer to a variable after you delete it, as a way to mark the pointer object as no longer pointing to anything.

```
int* ptr{new int{42}};
delete ptr;
ptr = nullptr;
```

A good programming practice is to ensure that no pointer variable ever retains an invalid value. Assigning nullptr to a pointer variable is one way to accomplish this.

A common idiom is to take advantage of short-circuiting (Exploration 12) to test for a null pointer, and if the pointer is not null, use it.

```
if (ptr != nullptr and ptr->needs_work())
    do_work(ptr);
```

A frequent question among newcomers to C++ is why the compiler does not generate code that automatically assigns a null pointer as part of the actions of the delete expression. There are two key reasons.

- The delete expression takes an rvalue as an argument, not necessarily a modifiable pointer. Thus, it may not have anything to modify.
- A program may have multiple pointers that all store the same address. Deleting the object invalidates all of these pointers. The delete expression cannot modify all of these pointers, because it doesn't know about them. It knows only about the one address that is the argument to the delete expression. Thus, modifying the argument to delete does not solve this problem. You can minimize the extent of this problem by not copying pointers.

C++ has some of its own uses for null pointers. You can supply a null pointer value to the delete expression, and it safely ignores the delete request. If you wish, you can ask that the new expression return a null pointer instead of throwing an exception when it cannot allocate enough memory for the new object. Just add (std::nothrow) after the new keyword and be sure to check the pointer that new returns. The parentheses are required, and std::nothrow is declared in <new>.

```
int *pointer{new (std::nothrow) int{42}};
if (pointer != nullptr)
  // do something with pointer...
```

Most programs don't require #include <new>. You need that header only if you use std::nothrow or catch std::bad_alloc.

If you define a pointer variable without initializing it, the variable's initial value is garbage. This is bad, and you are courting disaster—don't do it. If you don't have a valid address to use as the pointer's initial value, use nullptr.

Implementing Standard Containers

Have you ever wondered how the standard containers actually work? If, for instance, I were to ask you to implement std::map or std::set, could you do it? A full, high-quality implementation of any standard container is surprisingly difficult, but it isn't hard to grasp the basic principles.

The standard mandates logarithmic complexity for the associative containers. That is, lookups and insertion must have logarithmic performance, which pretty much forces a tree implementation. Most standard C++ libraries use red-black trees. A quick trip to the Internet will provide the algorithms—even working code—to implement red-black trees. The details of balancing the trees obscure the interesting use of pointers, so let's pick a simpler container: list.

The list class template implements a common doubly linked list. Start with a simplified definition of the list class template itself, as shown in Listing 59-1.

Listing 59-1. Defining the `list` Class Template

```
template<class T>
class list
{
public:
   list()
  : head_{nullptr}, tail_{nullptr}, size_{0}
  {}
  ~list() { clear(); }

  void clear();               ///< Erase all nodes in the list. Reset size to 0.
  void push_back(T const& x); ///< Add to tail.
  void pop_back();            ///< Erase from tail.
  // Other useful functions omitted for brevity...
private:
  class node
  {
  public:
    node(T const& key)
    : next_{nullptr}, prev_{nullptr}, value_{key}
    {}
    node* next_;
    node* prev_;
    T     value_;
  };
  node* head_;       ///< Start of the list
  node* tail_;       ///< End of the list
  std::size_t size_; ///< Number of nodes in the list
};
```

A list has a number of insert and erase functions. For the sake of simplicity, Listing 59-2 implements only push_back and pop_back.

Listing 59-2. Implementing `list::push_back` and `list::pop_back`

```
template<class T>
void list<T>::push_back(T const& x)
{
    node* n{new node{x}};
    if (tail_ == nullptr)
        head_ = tail_ = n;
    else
    {
        n->prev_ = tail_;
        tail_->next = n;
    }
    ++size_;
}

template<class T>
void list<T>::pop_back()
{
    node* n{tail_};
    if (head_ == tail_)
        head_ = tail_ = nullptr;
    else
    {
        tail_ = tail_->prev_;
        tail_->next_ = nullptr;
    }
    --size_;
    delete n;
}
```

Notice how pop_back removes the node from the list, ensures that the list is in a valid state, and then deletes the memory. Also notice how pop_back does not have to set n to a null pointer. Instead, the function simply returns; it has no opportunity to refer to the address of the deleted memory. These two techniques are ways that help ensure that your program handles dynamic memory correctly.

Adding Variables

Now return to the dependencies example that you started in Exploration 58. Let's add a new feature: variables. If an input line contains only one string, and the string contains an equal sign, it is a variable assignment. The variable name is to the left of the equal sign; the value is to the right. In this oversimplified example, no spaces are allowed around the equal sign or in the variable's value.

An artifact can contain a variable reference, which is a dollar sign, followed by the variable name in parentheses. The variable's value replaces the reference in its containing string. If a variable is not defined, automatically define it as an empty string.

```
NUM=1
target$(NUM) source$(NUM)
NUM=2
target$(NUM) source$(NUM)
source1 hdrX
source2 hdrY
all target1
all target2
```

The target all depends on target1 and target2. In turn, target1 depends on source1, and target2 depends on source2. Finally, source1 depends on hdrX, and source2 depends on hdrY.

Add an expand function that takes a string, expands variables, and returns the result. What if the expansion of a variable contains further variable references? I suggest expanding all variables and re-expanding the string until no more variables remain to be expanded. The std::string class has a number of helpful member functions: the find( ) function searches for the first occurrence of a substring or a character and returns the index of the substring or the constant std::string::npos to mean "not found." The index type is std::string::size_type. Pass an optional second argument to specify the position at which the search should begin.

```
std::string::size_type pos{str.find('$', 4)}; // start searching at index 4
```

The substr( ) member function takes two arguments, a starting position and a length, and returns a substring. The second argument is optional; omit it to mean "the rest of the string." The replace( ) function has several forms. It replaces a substring with a new string. Pass the starting index and length of the substring to replace, followed by the replacement string.

```
str.replace(start, length, replacement);
```

For now, use a global map to store variables. Call it variables. Listing 59-3 presents my implementation of the expand function.

Listing 59-3. Expanding Variables

```
typedef std::map<std::string, std::string> variable_map;
variable_map variables{};

std::string expand(std::string str)
{
   std::string::size_type start{0}; // start searching here
   while (true)
   {
      // Find a dollar sign.
      std::string::size_type pos{str.find('$', start)};
      if (pos == std::string::npos)
         // No more dollar signs.
         return str;
      if (pos == str.size() - 1 or str[pos + 1] != '(')
         // Not a variable reference. Skip the dollar sign,
         // and keep searching.
         start = pos + 1;
      else
      {
         std::string::size_type end{str.find(')', pos)};
         if (end == std::string::npos)
            // No closing parenthesis.
            return str;

         // Get the variable name.
         std::string varname{str.substr(pos + 2, end - pos - 2)};
         // Replace the entire variable reference.
         str.replace(pos, end - pos + 1, variables[varname]);
         // Scan the replacement text for more variables.
         start = pos;
      }
   }
}
```

Now modify the input function to recognize a variable definition, parse the definition to extract the variable name and value, and store the variable and its value in the variables map. Listing 59-4 illustrates how I rewrote the main function to parse variable definitions.

Listing 59-4. Parsing Variable Definitions

```
void parse_graph(std::istream& in, dependency_graph& graph)
{
  std::string line{};
  while (std::getline(in, line))
  {
    std::string target_name{}, dependency_name{};
    std::istringstream stream{line};
    if (stream >> target_name >> dependency_name)
    {
      artifact* target{lookup_artifact(expand(target_name))};
      artifact* dependency{lookup_artifact(expand(dependency_name))};
      graph.store_dependency(target, dependency);
    }
    else if (not target_name.empty())
    {
      std::string::size_type equal{target_name.find('=')};
      if (equal == std::string::npos)
        // Input line has a target with no dependency,
        // so report an error.
        std::cerr << "malformed input: target, " << target_name <<
                     ", must be followed by a dependency name\n";
      else
      {
        std::string variable_name{target_name.substr(0, equal)};
        std::string variable_value{target_name.substr(equal+1)};
        variables[variable_name] = variable_value;
      }
    }
    // else ignore blank lines
  }
}

int main()
{
  dependency_graph graph{};

  parse_graph(std::cin, graph);

  try {
    // Get the sorted artifacts in reverse order.
    std::vector<artifact*> sorted;
    graph.sort(std::back_inserter(sorted));

    // Then print the artifacts in the correct order.
    for (auto it(sorted.rbegin()), end(sorted.rend());
         it != end;
         ++it)
    {
      std::cout << (*it)->name() << '\n';
    }
  } catch (std::runtime_error const& ex) {
    std::cerr << ex.what() << '\n';
    return EXIT_FAILURE;
  }
}
```

This seemed like a good time to factor the input to its own function, parse_graph. So far, the modifications have not used dynamic memory, at least not explicitly. (Do you think the std::string class uses dynamic memory in its implementation?) The next step is to permit per-target variables. That is, if the input starts with a target name, and instead of a dependency name, the second string is a variable definition, that definition applies only to that target.

```
NUM=1
target$(NUM) SRC=1
target$(NUM) source$(SRC)
target2      source$(NUM)
target2      source$(SRC)
```

The NUM variable is global, so target2 depends on source1. The SRC variable applies only to target1, so target1 depends on source1. On the other hand, target2 depends on source, not source2, because target2 does not have a SRC variable, and unknown variables expand to an empty string.

One implementation is to add a map object to every artifact. Most artifacts do not have variables, however, so that can become wasteful. An alternative implementation uses dynamic memory and allocates a map only if a target has at least one variable. To look up a variable for a target, look only in the global map. To look up a variable for a dependency, first check the target's map then check the global map.

As the program evolves, the difference between a target and a dependency grows. This is to be expected, because in real life, they are quite different. Targets get actions, for example, so you can build them. You may well argue that now is a good time for refactoring, to create two derived classes: target and dependency. Only the target class could have a map for variables. I grant you extra credit if you decide to undertake this refactoring now. To keep the solution simple, however, I will make the fewest modifications possible as we go along.

Start with the new artifact class. In addition to the new map, add an expand member function, which hides the details of the two-level lookup. Compare your solution with mine, which is shown in Listing 59-5.

Listing 59-5. Adding Variable Storage and Lookup to the `artifact` Class

```
#ifndef ARTIFACT_HPP_
#define ARTIFACT_HPP_

#include <chrono>
#include <string>

#include "variables.hpp"

class artifact
{
public:
  typedef  std::chrono::system_clock clock;
  artifact()
  : name_{}, mod_time_{clock::time_point{}}, variables_{nullptr}
  {}
  artifact(std::string const& name)
  : name_{name}, mod_time_{get_mod_time()}, variables_{nullptr}
  {}
  ~artifact()
  {
    delete variables_;
  }

  std::string const& name()     const { return name_; }
  clock::time_point  mod_time() const { return mod_time_; }
  std::string        expand(std::string str) const
  {
    return ::expand(str, variables_);
  }

  /// Build a target.
  /// After completing the actions (not yet implemented),
  /// update the modification time.
  void build();

  /// Look up the modification time of the artifact.
  /// Return time_point{} if the artifact does not
  /// exist (and therefore must be built) or if the time cannot
  /// be obtained for any other reason.
  clock::time_point get_mod_time()
  {
    // Real programs should get this information from the
    // operating system. This program returns the current time.
    return clock::now();
  }
  /// Store a per-target variable.
  void store_variable(std::string const& name, std::string const& value)
  {
    if (variables_ == nullptr)
      variables_ = new variable_map;
    (*variables_)[name] = value;
  }
private:
  std::string name_;
  clock::time_point mod_time_;
  variable_map* variables_;
};

#endif // ARTIFACT_HPP_
```

Note that the new destructor does not have to test whether the variables_ variable is set to anything. The value will be either a null pointer or the address of a map. The delete expression handles both and does the right thing: nothing for a null pointer or deletes the memory for an address. Thus, the destructor is easy to write and understand.

I hid a number of details in the new header, variables.hpp, which I present in Listing 59-6.

Listing 59-6. The `variables.hpp` File

```
#ifndef VARIABLES_HPP_
#define VARIABLES_HPP_

#include <map>
#include <string>

typedef std::map<std::string, std::string> variable_map;
extern variable_map global_variables;

/// Expand variables in a string using a local map
/// and the global map.
/// @param str The string to expand
/// @param local_variables The optional, local map; can be null
/// @return The expanded string
std::string expand(std::string str, variable_map const* local_variables);

#endif // VARIABLES_HPP_
```

The implementation of the new expand function is in variables.cpp, shown in Listing 59-7.

Listing 59-7. The `variables.cpp` File Implements the expand Function

```
#include "variables.hpp"

variable_map global_variables{};

// Get a variable's value. Try the local variables first; if not found
// try the global variables. If still not found, define the name with
// an empty string and return an empty string. Subsequent lookups of
// the same name will find the empty string. Exercise for reader:
```

```
// print a message the first time the undefined variable's name
// is used.
std::string get_value(std::string const& name, variable_map const* local_variables)
{
   if (local_variables != nullptr)
   {
      variable_map::const_iterator iter{local_variables->find(name)};
      if (iter != local_variables->end())
         return iter->second;
   }
   return global_variables[name];
}

std::string expand(std::string str, variable_map const* local_variables)
{
   std::string::size_type start{0}; // start searching here
   while (true)
   {
      // Find a dollar sign.
      std::string::size_type pos{str.find('$', start)};
      if (pos == std::string::npos)
         // No more dollar signs.
         return str;
      if (pos == str.size() - 1 or str[pos + 1] != '(')
         // Not a variable reference.
         // Skip the dollar sign, and keep searching.
         start = pos + 1;
      else
      {
         std::string::size_type end{str.find(')', pos)};
         if (end == std::string::npos)
            // No closing parenthesis.
            return str;

         // Get the variable name.
         std::string varname{str.substr(pos + 2, end - pos - 2)};
         // Replace the entire variable reference.
         std::string value{get_value(varname, local_variables)};
         str.replace(pos, end - pos + 1, value);
         // Scan the replacement text for more variables.
         start = pos;
      }
   }
}
```

The only task that remains is to update the parser. **Modify the** parse_graph **function to parse target-specific variables.** Compare your solution with mine in Listing 59-8.

Listing 59-8. Adding per-Target Variables to parse_graph

```
void parse_graph(std::istream& in, dependency_graph& graph)
{
  std::string line{};
  while (std::getline(in, line))
  {
    std::string target_name{}, dependency_name{};
    std::istringstream stream{line};
    if (stream >> target_name >> dependency_name)
    {
      artifact* target{lookup_artifact(expand(target_name, 0))};
      std::string::size_type equal{dependency_name.find('=')};
      if (equal == std::string::npos)
      {
        // It's a dependency specification
        artifact* dependency{lookup_artifact(target->expand(dependency_name))};
        graph.store_dependency(target, dependency);
      }
      else
        // It's a target-specific variable
        target->store_variable(dependency_name.substr(0, equal-1),
                               dependency_name.substr(equal+1));
    }
    else if (not target_name.empty())
    {
      std::string::size_type equal{target_name.find('=')};
      if (equal == std::string::npos)
        // Input line has a target with no dependency,
        // so report an error.
        std::cerr << "malformed input: target, " << target_name <<
                     ", must be followed by a dependency name\n";
      else
        global_variables[target_name.substr(0, equal)] =
                                          target_name.substr(equal+1);
    }
    // else ignore blank lines
  }
}
```

Special Member Functions

The program seems to function, but it still needs work. As written, it triggers undefined behavior if the artifact class is copied. To understand what's going on, consider the program in Listing 59-9.

Listing 59-9. Simple Wrapper for Dynamic Memory

```
#include <iostream>

class wrapper
{
```

```
public:
   wrapper(int x) : p_{new int{x}} {}
   ~wrapper()                    { delete p_; }
   int value() const             { return *p_; }
private:
   int* p_;
};

void print(wrapper w)
{
   std::cout << w.value() << '\n';
}

wrapper increment(wrapper w)
{
   return wrapper(w.value() + 1);
}

int main()
{
  wrapper w{42};
  print(increment(w));
}
```

Predict the output from this program.

____

The behavior is undefined, so I cannot predict exactly how it will work on your system. Most likely, the program will fail, with some kind of memory violation. It is possible that the program will run without any observable sign of failure. It's also possible that it will print a seemingly random value instead of 43.

So what's going on? The constructor allocates an int; the destructor deletes an int; it all seems correct. The problem is the copy constructor.

"What copy constructor?" you ask.
"The copy constructor that the compiler creates for you," I answer.
"Oh, that copy constructor," you reply comprehendingly.

The compiler implicitly writes a copy constructor, assignment operator, destructor, and more for you, performing member-wise copying, assignment, and destruction (and more). In this case, the copy constructor dutifully copies the p_ data member. Thus, the original and the copy both contain identical pointers. The first one to be destroyed deletes the memory, leaving the other with a dangling pointer—an invalid pointer. When that other object is destroyed, it tries to delete p_ again, but it had already been deleted. Deleting the same address more than once is undefined behavior (unless a new expression has subsequently returned that same address).

One solution to this kind of problem is to disallow copying, as you learned in Exploration 33:

```
class nocopy {
public:
   nocopy(int x) : p_{new int{x}} {}
   ~nocopy()                     { delete p_; }
   nocopy(nocopy const&) = delete;
   void operator=(nocopy const&) = delete;
private:
   int* p_;
};
```

On the other hand, this means you can't copy or assign the objects, which is kind of limiting. Another solution is to make a *deep copy*—that is, allocate and copy the dynamic memory. Listing 59-10 shows this solution applied to Listing 59-9.

Listing 59-10. Making a Deep Copy

```
#include <iostream>

class wrapper
{
public:
   wrapper(int x)            : p_{new int{x}}         {}
   wrapper(wrapper const& w) : p_{new int{w.value()}} {}
   ~wrapper()                        { delete p_; }
   wrapper& operator=(wrapper w)
   {
      swap(w);
      return *this;
   }
   void swap(wrapper& w)
   {
      int* tmp{w.p_};
      w.p_ = p_;
      p_ = tmp;
   }
   int value() const                 { return *p_; }
private:
   int* p_;
};

void print(wrapper w)
{
   std::cout << w.value() << '\n';
}

wrapper increment(wrapper w)
{
   return wrapper{w.value() + 1};
}

int main()
{
  wrapper w{42};
  print(increment(w));
}
```

The implementation of the assignment operator is interesting, isn't it? You have a choice of many ways to implement this operator. The swap idiom has the advantage of simplicity. The standard library defines swap functions for the standard containers, and any decent library implements swap as a lightweight, fast function (if the container permits it). Typically, these swap functions work similar to wrapper::swap: they copy a few pointers. The trick of the assignment operator is that it takes its argument by value, not by reference. The compiler uses the copy constructor to make a copy of the source of the assignment; the swap function then swaps the current p_ value with the copy. The copy will be freed after the assignment operator returns, thereby cleaning up the original pointer, leaving the object with a deep copy of the assignment source, which is exactly what you want to have happen. It is not an optimal implementation, but it's easy and adequate, until you learn more in coming Explorations.

Deep copies are fine in some cases, but in others, they are wasteful. The pseudo-make program should never have to copy artifact objects. Each artifact object in the program represents a single artifact in the real world, and the real world doesn't magically make copies willy-nilly, so the program shouldn't either. Instead, you can move real objects from place to place. So why not move artifact objects in the program? That is the topic for the next Exploration.

EXPLORATION 60

Moving Data with Rvalue References

In Exploration 39, I introduced `std::move()` without explaining what it really does or how it works. Somehow it moves data, such as a string, from one variable to another, instead of copying the string contents, but you must be wondering how it works its wonders. The function itself is surprisingly simple. The concepts that underlie that simple implementation are more complicated, which is why I have waited until now to introduce the complexities and subtleties involved.

Temporary Objects

Historically, temporary objects could be the source of much unnecessary copying of data. For example, suppose you have a vector of strings, and you want to append a string to the end of the vector. A vector allocates an array in which to store its data (details forthcoming in Exploration 62). In C++ 03 and in a number of other languages, the vector class would ensure it has enough room and then copy the string into the vector. If the vector does not have enough room for another object, it must make room by allocating a new, larger array. It then copies all the strings out of the old array into the new.

One way to reduce the copying is to store only pointers in the vector. Copying a pointer is fast. But that requires a dynamic memory allocation for every object. Creating objects on the stack is much faster than using dynamic memory.

The goal, therefore, is to retain the benefit of creating objects on the stack, while eliminating unnecessary copying of data. When a vector has to grow, for example, the strings are moved into the new array without making any copies of the string content. When adding a string to the end of a vector, the string contents are moved into the vector without copying. Thus, it doesn't matter how large the strings are. Moving a big string is just as fast as moving a small string.

The trick is for the compiler to know when it is safe to move data and when it must make a copy. Ordinary assignment, for example, requires making a copy of the assignment source, so the target of the assignment ends up with an accurate copy of the source. Passing an argument to a function also requires making a copy of the argument, unless the parameter is declared as a reference type.

But other objects are temporary, and the compiler knows they are temporary. The compiler knows that it doesn't have to copy data out of the temporary object. The temporary is about to be destroyed; it doesn't need to retain its data, so the data can be moved to the destination without copying.

To help you visualize what happens when strings are copied, create a new class that wraps `std::string` and shows you when its copy constructor and assignment operator are called. Then create a trivial program that reads strings from std::cin and adds them to a vector. **Write the program** and compare your program with mine in Listing 60-1.

Listing 60-1. Exposing How Strings Are Copied

```
#include <iostream>
#include <string>
#include <vector>
```

```
class mystring : public std::string
{
public:
   mystring() : std::string{} { std::cout << "mystring()\n"; }
   mystring(mystring const& copy) : std::string(copy) {
      std::cout << "mystring copy(\"" << *this << "\")\n";
   }
};

std::vector<mystring> read_data()
{
   std::vector<mystring> strings{};
   mystring line{};
   while (std::getline(std::cin, line))
      strings.push_back(line);
   return strings;
}

int main()
{
   std::vector<mystring> strings{};
   strings = read_data();
}
```

(I changed the initialization style back to C++ 03, so you can try compiling and running the program with C++ 03 and C++ 11, just for fun. If you can't put your compiler into old-fashioned mode, that's okay. This Exploration is about C++ 11, and the historical excursion is just for fun.)

Try running the program with a few lines of input. **How many times is each string copied?** ______. The program copies line into the vector in push_back(). When the compiler returns the strings variable to the caller, it knows that it doesn't have to copy the vector. It can move it instead. Thus, you don't get any extra copies there (unless you tried the C++ 03 experiment, and you saw that the entire vector is copied).

How can we reduce the number of times the strings are copied? The line variable stores temporary data. The program has no reason to retain the value in line after calling push_back(). So we know it is safe to move the string contents out of line and into data. Call `std::move()` to tell the compiler that it can move the string into the vector. You also have to add a move constructor to mystring. See the new program in Listing 60-2. **Now how many times is each string copied?** ______.

Listing 60-2. Moving Strings Instead of Copying Them

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>

class mystring : public std::string
{
public:
   mystring() : std::string() { std::cout << "mystring()\n"; }
   mystring(mystring const& copy) : std::string(copy) {
      std::cout << "mystring copy(\"" << *this << "\")\n";
   }
   mystring(mystring&& move) noexcept
   : std::string(std::move(move)) {
      std::cout << "mystring move(\"" << *this << "\")\n";
   }
};

std::vector<mystring> read_data()
{
   std::vector<mystring> strings;
   mystring line;
   while (std::getline(std::cin, line))
      strings.push_back(std::move(line));
   return strings;
}

int main()
{
   std::vector<mystring> strings;
   strings = read_data();
}
```

The new constructor declares its parameters with a double ampersand (&&). It looks sort of like a reference. Notice that the parameter is not const. That's because moving data from an object must necessarily modify that object. Finally, recall from Exploration 45 that the noexcept specifier tells the compiler that the constructor cannot throw an exception. Notice also that the mystring constructor calls `std::move()` to move its parameter into the `std::string` constructor. You must call `std::move()` for any named object, even if that object is a special && reference.

The exact output depends on your library's implementation, but most start with a small amount of memory for the vector and grow slowly at first, to avoid wasting memory. Thus, adding just a few strings should show the vector reallocating its array. Table 60-1 shows the output from Listing 60-1, compiled in C++ 03 and C++ 11, as well as Listing 60-2, when I supply three lines of input.

Table 60-1. *Comparing Output of Listing 60-1 and Listing 60-2*

Listing 60-1 C++ 03	Listing 60-1 C++ 11	Listing 60-2 C++ 11
`mystring()`	`mystring()`	`mystring()`
`mystring copy("one")`	`mystring copy("one")`	`mystring move("one")`
`mystring copy("two")`	`mystring copy("two")`	`mystring move("two")`
`mystring copy("one")`	`mystring copy("one")`	`mystring move("one")`
`mystring copy("three")`	`mystring copy("three")`	`mystring move("three")`
`mystring copy("one")`	`mystring copy("two")`	`mystring move("two")`
`mystring copy("two")`	`mystring copy("one")`	`mystring move("one")`
`mystring copy("one")`		
`mystring copy("two")`		
`mystring copy("three")`		

The rest of this Exploration explains how C++ implements this move functionality.

Lvalues, Rvalues, and More

Recall from Exploration 21 that an expression falls into one of two categories: lvalue or rvalue. Informally, lvalues can appear on the left-hand side of an assignment, and rvalues appear on the right-hand side. Passing arguments to functions is similar to assignment: the function parameter takes on the role of lvalue, and the argument is an rvalue.

One key way to tell the difference between an lvalue and an rvalue is that you can take the address of an lvalue (using operator &). The compiler does not let you take the address of an rvalue, which makes sense. What is the address of 42?

The compiler automatically converts an lvalue to an rvalue whenever it has to, say, when passing an lvalue as an argument to a function or using an lvalue as the right-hand side of an assignment. The only situation in which the compiler turns an rvalue into an lvalue is when the lvalue's type is reference to const. For example, a function that declares its parameter as `std::string const&` can take an rvalue `std::string` as an argument, and the compiler turns that rvalue into an lvalue. But except for that one case, you cannot turn an rvalue into an lvalue.

The distinction between an lvalue and an rvalue is important when you consider the lifetime of an object. You know that the scope of a variable determines its lifetime, so any lvalue with a name (e.g., a variable or function parameter) has a lifetime determined by the name's scope. An rvalue, on the other hand, is temporary. Unless you bind a name to that rvalue (remember the name's type must be a reference to const), the compiler will destroy the temporary object as soon as it can.

For example, in the following expression, two temporary `std::string` objects are created and then passed to operator+ to concatenate the strings. The operator+ function binds its `std::string const&` parameters to the corresponding arguments, thereby guaranteeing that the arguments will live at least until the function returns. The operator+ function returns a new temporary `std::string`, which is then printed to std::cout:

```
std::cout << std::string("concat") + std::string("enate");
```

Once operator+ returns, the temporary `std::string` objects can be destroyed. The temporary `std::string` result from operator+ is then passed to operator<<, which binds its parameters to its arguments, ensuring that the string argument will not be destroyed until the function returns. After operator<< returns, the compiler is free to destroy the temporary `std::string` result of the concatenation.

The `std::move()` function lets you distinguish between the lifetime of an object and the data it contains, such as the characters that make up a string or the elements of a vector. The function takes an lvalue, whose lifetime is dictated by the scope, and turns it into an rvalue, so the contents can be treated as temporary. Thus, in Listing 60-1, the lifetime of line is determined by the scope. But in Listing 60-2, by calling `std::move()`, you are saying that it is safe to treat the string contents of line as temporary.

Because `std::move()` turns an lvalue into an rvalue, the return type (using the double ampersand) is called an *rvalue reference*. The parameters to the mystring move constructor also use double ampersands, so their types are rvalue references. A single-ampersand reference type is called an lvalue reference, to clearly distinguish it from rvalue references.

In a somewhat confusing turn of terminology, an expression with an rvalue reference type falls into both the rvalue expression category and the lvalue category. To reduce the confusion somewhat, a new name is given to this kind of expression: *xvalue*, for "eXpiring value." That is, the expression is still an lvalue and can appear on the left-hand side of an assignment, but it is also an rvalue, because it is near the end of its lifetime, so you are free to steal its contents.

A new name is given to rvalues that are not also xvalues: *pure rvalue*, or *prvalue*. Pure rvalues are expressions such as numeric literals, arithmetic expressions, function calls (if the return type is not a reference type), and so on. With a complete lack of symmetry, there are no pure lvalues. Instead, the term *lvalue* is used for the class of lvalues that are not also xvalues. The new term *generalized lvalue*, or *glvalue*, applies to all lvalues and xvalues. Figure 60-1 depicts the new expression categories.

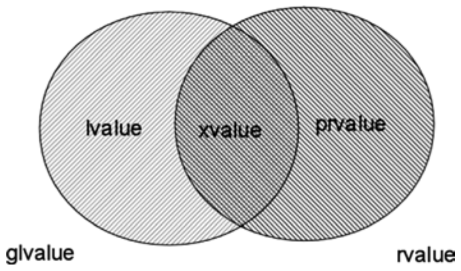


Figure 60-1. *Expression categories*

So it turns out `std::move()` is actually a trivial function. It takes an lvalue reference as an argument and turns it into an rvalue reference. The difference is important to the compiler in how it treats the expression, but `std::move()` does not generate any code and has no impact on performance.

To summarize:

- Calling a function that returns a type of lvalue reference returns an expression of category lvalue.
- Calling a function that returns a type of rvalue reference returns an expression of category xvalue.
- Calling a function that returns a non-reference type returns an expression of category prvalue.
- The compiler matches an rvalue (xvalue or prvalue) argument with a function parameter of type rvalue reference. It matches an lvalue argument with an lvalue reference.
- A named object has category lvalue, even if the object's type is rvalue reference.
- Declare a parameter with an rvalue reference type (using a double ampersand) to move data from the argument.
- Call `std::move()` as the source of an assignment or to pass an argument to a function when you want to move the data from an lvalue. This transforms an lvalue reference into an rvalue reference.

Implementing Move

Implementing a constructor for the mystring class is easy, because it simply moves its argument to its base class's move constructor. But how does a class such as `std::string` or `std::vector` implement move functionality? Go back to the artifact class in Listing 59-5. As you learned at the end of Exploration 59, copying an artifact leads to undefined behavior, due to the variables_ pointer. But it should be possible to move an artifact. Think about how you might go about moving an artifact. What, exactly, needs to be moved? How?

Moving the name is easy, because `std::string` already knows how to move itself. The tricky part is the variables_ member, which is a pointer. Okay, it's not tricky: a pointer has only one owner, so after moving it, set variables_ to nullptr in the move source. **Try writing a move constructor for the artifact class**. Then compare your solution with mine in Listing 60-3.

Listing 60-3. Adding a Move Constructor to the `artifact` Class

```
class artifact {
public:
   typedef std::chrono::system_clock clock;
   artifact(artifact&& source) noexcept
   : name_{std::move(source.name_)},
     mod_time_{std::move(source.mod_time_)},
     variables_{source.variables_}
   {
      source.variables_ = nullptr;
   }

   artifact& operator=(artifact&& source) noexcept
   {
      delete variables_;
      variables_ = source.variables_;
      source.variables_ = nullptr;
      name_ = std::move(source.name_);
      mod_time_ = std::move(source.mod_time_);
      return *this;
   }
   ... rest of artifact ...
};
```

Copy the variables_ pointer from the source to the destination, and then set the source pointer to nullptr. Thus, when the source is destroyed (which will happen soon), it will try to delete its variable map, but its pointer is now null. The constructor is a little easier, because you know you are starting from scratch. The assignment operator must deal with the possibility that variables_ already has a value. So delete the old variable map before moving the pointer from the source.

When you implement the move constructor and move assignment operator, be sure to supply the noexcept specifier. This tells the compiler that the function does not throw an exception. If your move functions are normal, all they do is copy some pointers, so the noexcept specifier is correct. The standard library needs to know that a move constructor does not throw an exception, so it can call that constructor from a container's own move constructors (which is also noexcept). The next Exploration will delve deeper into the subject of exceptions.

Rvalue or Lvalue?

All these xyzvalues can get confusing. To help you understand what's going on, Listing 60-4 shows a number of different expressions passed as arguments to overloaded `print()` functions.

Listing 60-4. Examining Expression Categories

```
#include <iostream>
#include <utility>

void print(std::string&& move)
{
   std::cout << "move: " << std::move(move) << '\n';
}
```

```
void print(std::string const& copy)
{
   std::cout << "copy: " << copy << '\n';
}

int main()
{
   std::string a{"a"}, b{"b"}, c{"c"};

  print(a);
  print(a + b);
  print(std::move(a));
  print(std::move(a + b));
  print(a + std::move(b));
  print(a + b + c);
}
```

Predict the output.

____

____

____

____

____

____

When I run the program, I get the following:

```
copy: a
move: ab
move: a
move: ab
move: ab
move: c
```

The precise output depends on your standard library. After `std::move`, the value of the argument is implementation-defined. Strings are often empty, but not necessarily so. Thus, your output may vary.

Special Member Functions

When you write a move constructor, you should also write a move assignment operator, and *vice versa*. You must also consider whether and how to write a copy constructor and copy assignment operator. The compiler will help you by implicitly writing default implementations or deleting the special member functions. This section takes a closer look at the compiler's implicit behavior and guidelines for writing your own special member functions (constructors, assignment operators, and destructor).

The compiler can implicitly create any or all of the following:

- default constructor, e.g., name()
- copy constructor, e.g., name(name const&)
- move constructor, e.g., name(name&&)
- copy assignment operator, e.g., name& operator=(name const&)
- move assignment operator, e.g., name& operator=(name&&)
- destructor, e.g., ~name()

A good guideline is that if you write any one of these special functions, you should write them all. You might decide that the compiler's implicit function is exactly what you want, in which case, you should say so explicitly with =default. That helps the human who maintains the code to know your intentions. If you know that the compiler would suppress a special member, note that explicitly with = delete.

As you know, the compiler deletes its implicit default constructor if you explicitly write any constructor. The implicit default constructor leaves pointers uninitialized, so if you have any pointer-type data member, you should write your own default constructor to initialize pointers to nullptr or delete the default constructor.

The compiler deletes its copy constructor and copy assignment operator if you explicitly provide a move constructor or move assignment operator. Thus, you know that copying an artifact class is unsafe. After implementing a move constructor, you don't want a copy constructor, and the compiler automatically obliges and suppresses the copy constructor and copy assignment operator.

The compiler deletes its move constructor if you explicitly provide any of the following special member functions: move assignment, copy constructor, copy assignment, or destructor.

The compiler deletes the move assignment operator if you explicitly provide any of the following special member functions: move constructor, copy constructor, copy assignment, or destructor.

The compiler's default behavior is to ensure safety. It will implicitly create copy and move functions if all the data members and base classes permit it. But if you start to write your own special members, the compiler assumes that you know best and suppresses anything that may be unsafe. It is then up to you to add back the special members that make sense for your class. When the compiler implicitly supplies any special member function, it also supplies the noexcept specifier when it is safe and correct to do so, that is, when all the data members and base classes also declare that function noexcept (or are built-in types).

Whenever a class allocates dynamic memory, you must consider all the special member functions. Ensure that pointer-type data members are always initialized. Ensure that move constructors and assignment operators correctly set the source pointers to nullptr. If you must implement a copy constructor or copy assignment operator, implement the appropriate deep copy or otherwise ensure that two objects are not both holding the same pointer. Make sure everything the class allocates gets deleted.

The requirements of data members apply to the class. Thus, if a data member lacks a copy constructor, then the compiler suppresses the implicit copy constructor for the containing class. After all, how could it copy the class if it can't copy all the members? Ditto for move functions.

As you can see, dynamic memory involves a number of complications. The next Exploration takes a look at more complications—namely, exceptions. Exceptions greatly complicate proper handling of dynamic memory, so pay close attention. (But not to worry, by Exploration 63, you will learn how to handle all these complications so that pointers will be easy to use.)

EXPLORATION 61

Exception-Safety

Exploration 45 introduced exceptions, which you have used in a number of programs since then. Dynamic memory presents a new wrinkle with regard to exceptions, and you must be that much more careful when handling them, in order to do so safely and properly in the face of dynamic memory management. In particular, you have to watch for memory leaks and similar problems.

Memory Leaks

Careless use of dynamic memory and exceptions can result in memory leaks—that is, memory that a program allocates but fails to free. In modern desktop operating systems, when an application terminates, the operating system reclaims all memory that the application used, so it is easy to become complacent about memory leaks. After all, no leak outlives the program invocation. But then your pesky users surprise you and leave your word processor (or whatever) running for days on end. They don't notice the memory leaking until suddenly they can no longer edit documents, and the automatic backup utility cannot allocate enough memory to save the user's work before the program terminates abruptly.

Maybe that's an extreme example, but leaking memory is a symptom of mismanaging memory. If you mismanage memory in one part of the program, you probably mismanage memory in other parts too. Those other parts may be less benign than a mere memory leak.

Consider the silly program in Listing 61-1.

Listing 61-1. Silly Program to Demonstrate Memory Leaks

```
#include <iostream>
#include <sstream>
#include <string>

int* read(std::istream& in)
{
  int value{};
  if (in >> value)
    return new int{value};
  else
    return nullptr;
}

int divide(int x, int y)
{
  return x / y;
}
```

```
int main()
{
  std::cout << "Enter pairs of numbers, and I will divide them.\n";
  std::string line{};
  while(std::getline(std::cin, line))
  {
    std::istringstream input{line};
    if (int* x{read(input)})
      if (int* y{read(input)})
        std::cout << divide(*x, *y) << '\n';
  }
}
```

This program introduces a new C++ feature. The if statements in main() define variables inside their conditionals. The rules for this feature are restrictive, so it is not used often. You can define only one declarator, and you must initialize it. The value is then converted to bool, which in this case means comparing to a null pointer. In other words, this conditional is true if the pointer is not null. The scope of the variable is limited to the body of the conditional (including the else portion of an if statement).

Now that you can understand it, **what's wrong with this program?**

____

____

____

The program leaks memory. It leaks memory if a line of text contains only one number. It also leaks memory if a line of text contains two numbers. In short, the program leaks like a termite's rowboat. Adding delete expressions should fix things, right? **Do it.** Your program may now look like Listing 61-2.

Listing 61-2. Adding `delete` Expressions to the Silly Program

```
#include <iostream>
#include <sstream>
#include <string>

int* read(std::istream& in)
{
  int value{};
  if (in >> value)
    return new int{value};
  else
    return nullptr;
}

int divide(int x, int y)
{
  return x / y;
}

int main()
{
  std::cout << "Enter pairs of numbers, and I will divide them.\n";
  std::string line{};
  while(std::getline(std::cin, line))
  {
    std::istringstream input{line};
    if (int* x{read(input)})
    {
      if (int* y{read(input)})
      {
        std::cout << divide(*x, *y) << '\n';
        delete y;
      }
      delete x;
    }
  }
}
```

Well, that's a little better, but only a little. Let's make the problem more interesting by adding some exceptions.

Exceptions and Dynamic Memory

Exceptions can be a significant factor for memory errors. You may write a function that carefully matches every new with a corresponding delete, but an exception thrown in the middle of the function will cause that oh-so-carefully-written function to fail, and the program forgets all about that dynamically allocated memory.

Anytime you use a new expression, you must be aware of places in your program that may throw an exception. You must have a plan for how to manage the exception, to ensure that you don't lose track of the dynamically allocated memory and that the pointer always holds a valid address. Many places can throw exceptions, including new expressions, any I/O statement (if the appropriate exception mask bit is set, as explained in Exploration 45), and a number of other library calls.

To see an example of how exceptions can cause problems, read the program in Listing 61-3.

Listing 61-3. Demonstrating Issues with Exceptions and Dynamic Memory

```
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>

int* read(std::istream& in)
{
  int value{};
  if (in >> value)
    return new int{value};
  else
    return nullptr;
}

int divide(int x, int y)
{
  if (y == 0)
    throw std::runtime_error{"integer divide by zero"};
  else if (x < y)
    throw std::underflow_error{"result is less than 1"};
  else
    return x / y;
}

int main()
{
  std::cout << "Enter pairs of numbers, and I will divide them.\n";
  std::string line{};
  while(std::getline(std::cin, line))
    try
    {
      std::istringstream input{line};
      if (int* x{read(input)})
      {
        if (int* y{read(input)})
        {
          std::cout << divide(*x, *y) << '\n';
          delete y;
        }
        delete x;
      }
    } catch (std::exception const& ex) {
      std::cout << ex.what() << '\n';
    }
}
```

Now what's wrong with this program?

____

____

____

The program leaks memory when the divide function throws an exception. In this case, the problem is easy to see, but in a more complicated program, it can be harder to identify. Looking at the input loop, it seems that every allocation is properly paired with a delete expression. But in a more complicated program, the source of exceptions and the try-catch statement may be far apart and unrelated to the input loop.

Ideally, you should be able to manage memory without knowing about exceptions. Fortunately, you can—at least to a certain degree.

Automatically Deleting Pointers

Keeping track of allocated memory can be tricky, so you should accept any help that C++ can offer. One class template that can help a lot is std::unique_ptr<> (defined in the <memory> header). This template wraps a pointer, so that when the unique_ptr object goes out of scope, it automatically deletes the pointer it wraps. The template also guarantees that exactly one unique_ptr object owns a particular pointer. Thus, when you assign one unique_ptr to another, you know exactly which unique_ptr (the target of the assignment) owns the pointer and has responsibility for freeing it. You can assign unique_ptr objects, pass them to functions, and return them from functions. In all cases, ownership passes from one unique_ptr object to another. Like children playing the game of Hot Potato, whoever is left holding the pointer or potato in the end is the loser and must delete the pointer.

The unique_ptr template is particularly helpful when a program throws an exception. When C++ handles the exception, it unwinds the stack and destroys local variables along the way. This means it will destroy local unique_ptr objects in those unwound stack frames, which will delete their pointers. Without unique_ptr, you may get a memory leak.

Thus, a common idiom is to use unique_ptr<> for local variables of pointer type and for data members of pointer type. Equally viable, but less common, is for function parameters and return types to be unique_ptr. Return a unique_ptr normally, but a function parameter must be declared as an rvalue reference, and the caller must move the pointer to the function, as illustrated in Listing 61-4.

Listing 61-4. Using the `unique_ptr` Class Template

```
#include <iostream>
#include <memory>

class see_me
{
public:
  see_me(int x) : x_{x} { std::cout <<  "see_me(" << x_ << ")\n"; }
  ~see_me()             { std::cout << "~see_me(" << x_ << ")\n"; }
  int value() const     { return x_; }
private:
  int x_;
};

std::unique_ptr<see_me> nothing(std::unique_ptr<see_me>&& arg)
{
  return std::move(arg);
}

template<class T>
std::unique_ptr<T> make(int x)
{
  return std::unique_ptr<T>{new T{x}};
}

int main()
{
  std::cout << "program begin...\n";
  std::unique_ptr<see_me> sm{make<see_me>(42)};
  std::unique_ptr<see_me> other;
  other = nothing(std::move(sm));
  if (sm.get() == nullptr)
    std::cout << "sm is null, having lost ownership of its pointer\n";
  if (other.get() != nullptr)
    std::cout << "other now has ownership of the int, " <<
                 other->value() << '\n';
  std::cout << "program ends...\n";
}
```

As you can see, the get() member function returns the raw pointer value, which you can use to test the pointer or pass to functions that do not expect to gain ownership of the pointer. You can assign a new unique_ptr value, which causes the target of the assignment to delete its old value and take ownership of the new pointer. Dereference the unique_ptr pointer with *, or use -> to access members, the same way you would with an ordinary pointer.

Use std::unique_ptr to fix the program in Listing 61-3. Compare your repairs with mine, which are presented in Listing 61-5.

Listing 61-5. Fixing Memory Leaks

```
#include <iostream>
#include <memory>
#include <sstream>
#include <stdexcept>
#include <string>

std::unique_ptr<int> read(std::istream& in)
{
  int value;
  if (in >> value)
    return std::unique_ptr<int>{new int{value}};
  else
    return std::unique_ptr<int>{};
}

int divide(int x, int y)
{
  if (y == 0)
    throw std::runtime_error("integer divide by zero");
  else if (x < y)
    throw std::underflow_error("result is less than 1");
  else
    return x / y;
}

int main()
{
  std::cout << "Enter pairs of numbers, and I will divide them.\n";
  std::string line{};
  while(std::getline(std::cin, line))
    try
    {
      std::istringstream input{line};
      if (std::unique_ptr<int> x{read(input)})
      {
        if (std::unique_ptr<int> y{read(input)})
          std::cout << divide(*x, *y) << '\n';
      }
    } catch (std::exception const& ex) {
      std::cout << ex.what() << '\n';
    }
}
```

The changes are minimal, but they vastly increase the safety of this program. No matter what happens in the divide function or elsewhere, this program does not leak any memory. You will learn more about unique_ptr in Exploration 63.

Exceptions and Constructors

Even without unique_ptr, C++ guarantees one level of exception-safety when constructing an object: if a constructor throws an exception, the compiler automatically cleans up base-class portions of the incompletely constructed object. If some data members have been initialized successfully, they are destroyed in reverse order of creation, but uninitialized data members are left alone. The new expression never completes, so it never returns a pointer to an incomplete object. But a raw pointer data member has no destructor. The compiler does not automatically delete the memory, so any memory allocated for pointer-type data members will be stranded. Listing 61-6 demonstrates how constructors and exceptions interact.

Listing 61-6. Demonstrating Constructors That Throw Exceptions

```
#include <iostream>

class see_me
{
public:
  see_me(int x) : x_{x} { std::cout <<  "see_me(" << x_ << ")\n"; }
  ~see_me()             { std::cout << "~see_me(" << x_ << ")\n"; }
private:
  int x_;
};

class bomb : public see_me
{
public:
  bomb() : see_me{1}, a_{new see_me{2}} { throw 0; }
  ~bomb() {
    delete a_;
  }
private:
  see_me *a_;
};

int main()
{
  bomb *b{nullptr};
  try {
    b = new bomb;
  } catch(int) {
    if (b == nullptr)
      std::cout << "b is null\n";
  }
}
```

Predict the output from this program:

____

____

____

____

____

Run the program. **What is the actual output?**

____

____

____

____

____

Explain your observations.

____

____

____

____

____

The bomb class throws an exception in its constructor. It derives from see_me and has an additional member of type pointer to see_me. The see_me class lets you see the constructors and destructors, so you can see that the data member, see_me(2), is constructed, but never destroyed, which indicates a memory leak. The bomb destructor never runs, because the bomb constructor never finishes. Therefore, a_ is never cleaned up. On the other hand, see_me(1) is cleaned up because the base class is automatically cleaned up if a derived-class constructor throws an exception.

The main() function catches the exception and recognizes that the b variable was never assigned. Thus, there's nothing for the main() function to clean up.

What happens if you were to use unique_ptr in the bomb class? **Try and see. What happens?**

____

____

____

Listing 61-7 shows the new program.

Listing 61-7. Using `unique_ptr` in `bomb`

```
#include <iostream>
#include <memory>

class see_me
{
```

```
public:
  see_me(int x) : x_{x} { std::cout <<  "see_me(" << x_ << ")\n"; }
  ~see_me()             { std::cout << "~see_me(" << x_ << ")\n"; }
private:
  int x_;
};

class bomb : public see_me
{
public:
  bomb() : see_me{1}, a_{new see_me{2}} { throw 0; }
  ~bomb() {}
private:
  std::unique_ptr<see_me> a_;
};

int main()
{
  bomb *b{nullptr};
  try {
    b = new bomb;
  } catch(int) {
    if (b == nullptr)
      std::cout << "b is null\n";
  }
}
```

Notice that all the see_me objects are now properly destroyed. Even though the constructor does not finish before throwing an exception, any data members that have been constructed will be destroyed. Thus, unique_ptr objects are cleaned up. Ta da! Mission accomplished!

Or is it? Even with unique_ptr, you must still be cautious. Consider the program in Listing 61-8.

Listing 61-8. Mystery Program That Uses `unique_ptr`

```
#include <iostream>
#include <memory>

class mystery
{
public:
  mystery() {}
  mystery(mystery const&) { throw "oops"; }
};

class demo
{
public:
  demo(int* x, mystery m, int* y) : x_{x}, m_{m}, y_{y} {}
  demo(demo const&) = delete;
  demo& operator=(demo const&) = delete;
  int x() const { return *x_; }
  int y() const { return *y_; }
private:
  std::unique_ptr<int> x_;
  mystery            m_;
  std::unique_ptr<int> y_;
};

int main()
{
  demo d{new int{42}, mystery{}, new int{24}};
  std::cout << d.x() << d.y() << '\n';
}
```

What's wrong with this program?

____

____

____

To help you understand what the program does, use a see_me object instead of int. Does that help you understand?

The demo class uses unique_ptr to ensure proper lifetime management of its pointers. It deletes its copy constructor and copy assignment operator to avoid any problems they may cause.

The problem is the basic design of the demo constructor. By taking two pointer arguments, it opens the possibility of losing track of these pointers before it can safely tuck them away in their unique_ptr wrappers. The mystery class forces an exception, but in a real program, unexpected exceptions can arise from a variety of less explicit sources.

The simplest solution is to force the caller to use unique_ptr by changing the demo constructor, as demonstrated in the following:

```
demo(std::unique_ptr<int>&& x, mystery m, std::unique_ptr<int>&& y)
: x_{std::move(x)}, m_{m}, y_{std::move(y)}
{}
```

The unique_ptr class would have simplified much of the messiness of dealing with pointers back in Exploration 59, but if you learned about unique_ptr first, you wouldn't be able to fully appreciate all that it does for you. It has even more magic to help you, and Exploration 63 will take a closer look at unique_ptr and some of its friends. But first, let's take a side trip and discover the close connection between old-fashioned, C-style arrays and pointers.

EXPLORATION 62

Old-Fashioned Arrays

Throughout this book, I've used std::vector for arrays. Exploration 53 touched on std::array and introduced other containers, such as std::deque. Hidden in the implementation of these types is an old-fashioned, crude, and unsafe style of array. This Exploration takes a look at this relic from C, not because I want you ever to use it, but because you may have to read code that uses this language construct, and it will help you to understand how the standard library can implement vector, array, and similar types. You may be surprised to learn that C-style arrays have much in common with pointers.

C-Style Arrays

C++ inherits from C a primitive form of array. Although an application should never have to use C-style arrays, library authors sometimes have to use them. For example, a typical implementation of std::vector makes use of C-style arrays.

The following shows how you define a C-style array object by specifying the array size in square brackets after the declarator name:

```
int data[10];
```

The array size must be a compile-time constant integer expression. The size must be strictly positive; zero-length arrays are not allowed. The compiler sets aside a single chunk of memory that is large enough to store the entire array. After the array definition, your code can use the array name as an address, not as a pointer. The elements of the array are lvalues, so, for instance, you can assign to data[0] but not to data itself.

Use square brackets to refer to elements of the array. The array index must be an integer. If the index is out of bounds, the results are undefined. Now you can see why std::vector implements the square bracket operator the way it does—namely, in imitation of C-style arrays.

```
int data[10];
std::vector<int> safer_data{10};
data[0] = 42;           // okay
safer_data[0] = 42;     // okay: just like a C-style array
safer_data.at(0) = 42;  // okay: safer way to access vector elements
data[10] = -1;          // error: undefined behavior
safer_data[10] = -1;    // error: also undefined behavior
safer_data.at(10) = -1; // okay: throws an exception
```

To initialize an array, use curly braces, just like universal initialization.

```
int data[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
point corners[2] { point{0, 1}, point{10, 20} };
```

When you provide initial values, you can omit the array size. The compiler uses the number of initializers as the array size.

```
int data[] { 1, 2, 3, 4, 5 }; // just like data[5]
```

If you provide an array size, you can omit trailing elements, in which case the compiler zero-initializes the remaining elements, but only if the declarator has an initializer. With no initializer, the array elements are default-initialized, that is, builtin types are uninitialized.

```
int values[5]{ 3, 2, 1 }; // like { 3, 2, 1, 0, 0 }
int data[10]{ };          // initialize entire array to zero
```

Array Limitations

One of the key limitations to a C-style array is that the array doesn't know its own size. The compiler knows the size when it compiles the array definition, but the size is not stored with the array itself, so most uses of the array are not aware of the array size.

If you declare an array type as a function parameter, something strange happens: the compiler ignores the size and treats the array type as a pointer type. In other words, when used as a function parameter, int x[10] means exactly the same thing as int x[1], int x[100000000], and int* x. In practical terms, this means the function has no idea what the array size is. The function sees only a pointer to the start of the array. Thus, functions that take a C-style array as an argument typically have an additional argument to pass the array size, as shown in Listing 62-1.

Listing 62-1. Array Type in Function Parameters

```
#include <iostream>

int sum(int* array, int size);

int main()
{
  int data[5]{ 1, 2, 3, 4, 5 };
  std::cout << sum(data, 5) << '\n';
}

int sum(int array[], int size)
{
  int result{0};
  while (size-- != 0)
    result += array[size];
  return result;
}
```

Because an array does not store its size, an extern declaration (Exploration 41) of an array doesn't keep track of the array size. Thus, the definition of the array must specify the size (explicitly or implicitly with an initializer), but extern declarations can omit the size. Unlike function arguments and parameters, no conversions take place with extern objects. Arrays are arrays, and pointers are pointers. If the extern declaration does not match the object's definition, the results are undefined.

Another limitation of arrays is that a function cannot return an array.

Dynamically Allocating an Array

A new expression can allocate an array, or more precisely, it allocates one or more contiguous values, default-initializes each element, and returns the address of the first one. (Remember that default-initializing a built-in type, such as int, leaves the value uninitialized.) Initialize elements of the array with values in curly braces. Remember that omitted elements are zero-initialized, so {} initializes every element of the new array. Like passing an array to a function, all you get from new is a pointer. It is up to you to keep track of the size. Pass the size in square brackets after the type.

```
int* ptr{ new int[10]{ 1, 2, 3 } };
ptr[0] = 9;
ptr[9] = 0;
```

To free the memory, use the delete[] operator. The square brackets are required. Do not pass the size.

```
delete[] ptr;
```

If you allocate a scalar value with new (no square brackets), you must delete the memory with plain delete (no square brackets). If you allocate an array (even if the size is one), delete the memory with delete[]. You cannot mix the array-style new with a non-array delete or vice versa. Because the delete operator and delete[] operator both take a plain pointer as an operand, the compiler cannot, in general, detect errors. A good library can detect errors and report them at runtime, but the standard provides no guarantee.

The unique_ptr type distinguishes between scalars and arrays, so you can use it to manage dynamically allocated arrays, as follows:

```
std::unique_ptr<int[]> array{ new int[10] };
std::unique_ptr<int> scalar{ new int };
```

In both cases, the expression type is int*, so there is no way for the compiler to verify that you are using the correct form of unique_ptr with a matching new expression. Enable all the debugging features in your compiler and see if it detects any errors in Listing 62-2.

Listing 62-2. Deliberate Errors with new and delete

```
#include <memory>

int main()
{
   int* p{ new int[10] };
   delete p;
   p = new int;
   delete[] p;
   std::unique_ptr<int> up{ new int[10] };
   up.reset();
   std::unique_ptr<int[]> upa{ new int };
   upa.reset();
}
```

The array Type

Many of the limitations of C-style arrays are ameliorated by std::array. Unlike C-style arrays, std::array remembers its size. Unlike C-style arrays, a function can return std::array. Unlike C-style arrays, std::array offers bounds-checking with the at() member function. Like std::vector, you must provide an element type as a template argument, but you also supply the array size. Also, std::array can have zero size, but C-style arrays cannot. One oddity is that an array initializer requires a second set of curly braces, which is a side effect of the way std::array works.

```
std::array<int, 5> data{ { 1, 2, 3, 4, 5 } };
assert(data.size() == 5);
data.at(10); // throws an exception
for (auto x : data)  // supports iterators
    std::cout << x << '\n';
std::array<int, 0> look_ma_no_elements;
assert(look_ma_no_elements.size() == 0);
```

But C-style arrays offer one (only one!) feature lacking in std::array. **What is it?**

____

The compiler can count the number of initializers of a C-style array, but you must do the counting for std::array. There are some situations in which this limitation is serious, but in most cases, std::array is vastly superior to C-style arrays.

Multidimensional Arrays

All C-style arrays are one-dimensional, but you can create an array of arrays. For example, define a variable as a 3 × 4 matrix as follows:

```
double matrix[3][4];
```

Read this declaration the way you would any other. Start with the name and work your way from inside to outside: matrix is an array with three elements. Each element is an array of four elements of type double. Thus, C++ arrays are row-major—that is, the array is accessed by row, and the right-most index varies fastest. So another way to define matrix is as follows:

```
typedef double row[4];
row matrix[3];
```

When you pass a matrix to a function, only the left-most array is converted to a pointer. Thus, if you were to pass matrix to a function, you would have to declare the function parameter as a pointer to an array of four doubles, as follows:

```
double sum(double arg[][4]);
```

or:

```
double sum(double *arg[4]);
```

or:

```
double sum(row* arg);
```

To refer to elements of the matrix, use a separate subscript operator for each index, as follows:

```
void initialize(double matrix[][4], int num_rows)
{
   for (int i = 0; i != num_rows; ++i)
      for (int j = 0; j != 4; ++j)
         matrix[i][j] = 1.0;
}
```

You can also refer to an entire row: matrix[2] returns the address of the last row of the matrix, which has type double[4], which means it is the address of the first element of a four-element array of double.

If you use std::array, you can treat the matrix as an object, with all the advantages of std::array. The declaration syntax might strike you as backward, but it clearly expresses the nature of a multidimensional C array as an array of arrays, as in the following:

```
std::array<std::array<double, 4>, 3> matrix;
```

C-Style Strings

Another legacy type that C++ inherits from C is the C-style string, which is little more than a C-style array of char. (A wide C-style string is a C-style array of wchar_t. Everything in this Exploration applies equally to wide strings and wchar_t but mentions only char for the sake of simplicity. Ditto for strings of char16_t and char32_t.) A string literal in C++ is a const array of char. The size of the array is the number of elements in the array, plus one. The compiler automatically appends the character with value zero ('\0') to the end of the array, as a marker for the end of the string. (Remember that the array does not store the size, so the trailing zero-valued character is the only way to identify the end of the string and, therefore, the length of the string.) The zero-value character is also called a *null character*. In spite of the unfortunate collision of terminology, null characters have nothing to do with null pointers (Exploration 59).

The std::string class has a constructor to construct a C++ string from a C character pointer. Often, the compiler is able to call this constructor automatically, so you can usually use a string literal anywhere that calls for std::string.

Should you ever have to work with C-style strings directly, remember that a string literal contains const elements. A frequent mistake is to treat a string literal as an array of char, not an array of const char. Although you generally cannot know the amount of memory that a character array occupies, you can discover the number of characters (storage units, not logical characters, if the character set is multi-byte) by calling the std::strlen function (declared in <cstring>, along with several other functions that are useful for working with C-style strings), passing the start of the character array as an argument. (Wide characters have different support functions; for details consult a library reference.)

Command-Line Arguments

The one and only time you should use a C-style array is to access command-line arguments that the host environment passes to a program when the main() function begins. For historic reasons, the command-line arguments are passed as a C-style array of pointers to C-style character arrays. Thus, you can choose to write the main() function as a function of no arguments or a function of two arguments: an int for the number of command-line arguments and a pointer to the first element of an array of pointers to the individual command line arguments, each as an array of char (not const char). Listing 62-3 shows an example of echo, which echoes command-line arguments to the standard output. Note that the first command-line argument is the program name or path to the program's executable file (the details are defined by the implementation). Note also that std::ostream knows how to print a C-style character pointer by printing the character contents of the string.

Listing 62-3. Echoing Command-Line Arguments

```
#include <iostream>

int main(int argc, char* argv[])
{
  char const* separator{""};
  while (--argc != 0)
  {
    std::cout << separator << *++argv;
    separator = " ";
  }
}
```

The names argc and argv are conventional, not required. As with any other function parameters, you are free to pick any names you want. The second argument is of type pointer-to-pointer-to-char and is often written as char* argv[] to emphasize the point that it is an array of char* values, although some programmers also use char** argv, which means the same thing.

The size of the argv array is argc + 1, because its last element is a null pointer, after all the command-line arguments. Thus, some programs loop through command-line arguments by counting and comparing with argc, and others loop through argv until reaching a null pointer.

Write a program that takes two command-line arguments: an input file and an output file. The program copies the contents of the input file to the output file. Compare your solution with mine, shown in Listing 62-4.

Listing 62-4. Copying a File Named on the Command Line

```
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>

int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    std::cerr << "usage: " << argv[0] << " INPUT OUTPUT\n";
    return EXIT_FAILURE;
  }
  std::ifstream input{argv[1]};
  if (not input)
  {
    std::perror(argv[1]);
    return EXIT_FAILURE;
  }
  std::ofstream output{argv[2]};
  if (not output)
  {
    std::perror(argv[2]);
    return EXIT_FAILURE;
  }

  input.exceptions(input.badbit);    // throw for serious errors
  output.exceptions(output.failbit); // throw for any error

  try
  {
    // Lots of ways to copy: use std::copy, use a loop to read & write
    // The following is a little-known technique that is probably fastest.
    output << input.rdbuf();
    output.close();
    input.close();
  }
  catch (std::ios_base::failure const& ex)
  {
    std::cerr << "Can't copy " << argv[1] << " to " << argv[2] << ": " <<
              ex.what() << '\n';
    return EXIT_FAILURE;
  }
}
```

Pointer Arithmetic

An unusual feature of C++ pointers (inherited from C) is that you can perform addition and subtraction on pointers. In ordinary usage, these operations work only on pointers that point into arrays. Specifically, you can add or subtract integers and pointers, and you can subtract two pointers to get an integer. You can also compare two pointers, using relational operators (less than, greater than, etc.). This section explores what these operations mean.

Briefly, a pointer can point to any object in an array. Add an integer to a pointer to obtain the address of an element of the array. For example, array + 2 points to the third element of the array: the element at index 2. You are allowed to form a pointer to any position in the array, including the position one past the end. Given a pointer into the array, subtract an integer to obtain the address of an element earlier in the array. You are not allowed to form an address that precedes the first element of the array.

Subtract two pointers to obtain the number of array elements that separate them. They must be pointers that point into the same array. When you compare two pointers using relational operators, pointer a is "less than" pointer b if a and b both point to the same array and a comes earlier in the array than b.

Ordinarily, you have no reason to use the relational operators on pointers that are not in the same array. But you can use pointers as keys in sets and maps, and these types have to compare pointers to put the keys in order. The std::set and std::map templates use std::less to compare keys (Exploration 50), and std::less uses the < operator. The details are specific to the implementation, but the standard requires std::less to work with all pointers, thereby ensuring that sets and maps work properly when you use pointers as keys.

The compiler and library are not required to enforce the rule that pointers must point to the same array or stay confined to legal indices. Some compilers might try to give you a few warnings, but in general, the compiler cannot tell whether your program follows all the rules. If your program does not follow the rules, it enters the twilight zone of undefined behavior. That's what makes pointers so dangerous: it's easy to fall into undefined behavior territory.

The most common use for pointer arithmetic is to advance through the elements of an array by marching a pointer from the beginning of the array to the end, instead of using an array index. Listing 62-5 illustrates this idiom as well as pointer subtraction, by showing one possible implementation of the standard std::strlen function, which returns the length of a C-style string.

Listing 62-5. Using Pointer Arithmetic to Determine the Length of a C String

```
#include <cstddef>

std::size_t my_std_strlen(char const* str)
{
   char const* start{str};      // remember the start of the string
   while (*str != 0)            // while not at the end of the string
      ++str;                    // advance to the next character
   return str - start;          // compute string length by subtracting pointers
}
```

Pointer arithmetic is error-prone, dangerous, and I recommend avoiding it. Instead of C strings, for example, use std::string. Instead of C-style arrays, use std::vector or std::array.

However, pointer arithmetic is a common idiom in C++ programs and, therefore, unavoidable. Pointer arithmetic is especially prevalent in library implementations. For example, I can almost guarantee that it is used in your library's implementation of the string, vector, and array class templates. Thus, library authors must be especially vigilant against errors that are difficult or impossible for the compiler to detect, but that effort pays off by making a safer interface available to all other developers.

In the interest of making pointers safer, C++ lets you define a class that looks, acts, and smells like a pointer type but with bonus features, such as additional checks and safety. These so-called smart pointers are the subjects of the next Exploration.

EXPLORATION 63

Smart Pointers

The std::unique_ptr class template is an example of a so-called *smart pointer*. A smart pointer behaves much like any other pointer but with extra features and functionality. This Exploration takes a closer look at unique_ptr and other smart pointers.

Revisiting unique_ptr

Exploration 61 introduced unique_ptr as a way to manage dynamically allocated objects. The unique_ptr class template overloads the dereference (*) and member access (->) operators and lets you use a unique_ptr object the same way you would use a pointer. At the same time, it extends the behavior of an ordinary pointer, such that when the unique_ptr object is destroyed, it automatically deletes the pointer it holds. That's why unique_ptr is called a *smart pointer*—it's just like an ordinary pointer, only smarter. Using unique_ptr helps ensure that memory is properly managed, even in the face of unexpected exceptions.

When used properly, the key feature of unique_ptr is that exactly one unique_ptr object owns a particular pointer. You can move unique_ptr objects. Each time you do, the target of the move becomes the new owner of the pointer.

You can also force a unique_ptr to give up ownership of its pointer by calling the release() member function. The release() function returns the raw pointer, as displayed in the following:

```
std::unique_ptr<int> ap{new int{42}};
int* ip{ap.release()};
delete ip;
```

Call the reset member function to tell a unique_ptr to take over a different pointer. The unique_ptr object takes control of the new pointer and deletes its old pointer. With no argument, reset() sets the unique_ptr to a null pointer.

```
std::unique_ptr<int> ap{new int{42}};
ap.reset(new int{10}); // deletes the pointer to 42
ap.reset();            // deletes the pointer to 10
```

The get() member function retrieves the raw pointer without affecting the unique_ptr's ownership. The unique_ptr template also overloads the dereference (*) and member (->) operators, so that they work the way they do with ordinary pointers. These functions do not affect ownership of the pointer.

```
std::unique_ptr<rational> rp{new rational{420, 10}};
int n{rp->numerator()};
rational r{*rp};
rational *raw_ptr{rp.get()};
```

When unique_ptr holds a pointer to an array (that is, the template argument is an array type, e.g., unique_ptr<int[]>), it supports the subscript operator instead of * and ->.

In order to enforce its ownership semantics, unique_ptr has a move constructor and move assignment operator but deletes its copy constructor and copy assignment operator. If you use unique_ptr for data members in a class, the compiler implicitly deletes the class's copy constructor and copy assignment operator.

Thus, using unique_ptr may free you from thinking about your class's destructor, but you are not excused from thinking about the constructors and assignment operators. This is a minor tweak to the guideline that if you have to deal with one, you must deal with all special members. The compiler's default behavior is usually correct, but you might want to implement a copy constructor that performs a deep copy or other non-default behavior.

Copyable Smart Pointers

Sometimes, you don't want exclusive ownership. There are circumstances when multiple objects will share ownership of a pointer. When no objects own the pointer, the memory is automatically reclaimed. The std::shared_ptr smart-pointer type implements shared ownership.

Once you deliver a pointer to a shared_ptr, the shared_ptr object owns that pointer. When the shared_ptr object is destroyed, it will delete the pointer. The difference between shared_ptr and unique_ptr is that you can freely copy and assign shared_ptr objects with normal semantics. Unlike unique_ptr, shared_ptr has a copy constructor and copy assignment operator. The shared_ptr object keeps a reference count, so assignment merely increments the reference count, without having to transfer ownership. When a shared_ptr object is destroyed, it decrements the reference count. When the count reaches zero, the pointer is deleted. Thus, you can make as many copies as you like, store shared_ptr objects in a container, pass them to functions, return them from functions, copy them, move them, assign them, and carry on to your heart's content. It's that simple. Listing 63-1 shows that copying shared_ptr works in ways that don't work with unique_ptr.

Listing 63-1. Working with `shared_ptr`

```
#include <iostream>
#include <memory>
#include <vector>

class see_me
{
public:
  see_me(int x) : x_{x} { std::cout <<  "see_me(" << x_ << ")\n"; }
  ~see_me()             { std::cout << "~see_me(" << x_ << ")\n"; }
  int value() const     { return x_; }
private:
  int x_;
};

std::shared_ptr<see_me> does_this_work(std::shared_ptr<see_me> x)
{
  std::shared_ptr<see_me> y{x};
  return y;
}

int main()
{
  std::shared_ptr<see_me> a{}, b{};
  a = std::make_shared<see_me>(42);
  b = does_this_work(a);
  std::vector<std::shared_ptr<see_me>> v{};
  v.push_back(a);
  v.push_back(b);
}
```

The best way to create a shared_ptr is to call make_shared. The template argument is the type you want to create, and the function arguments are passed directly to the constructor. Due to implementation details, constructing a new shared_ptr instance any other way is slightly less efficient in space and time.

Using shared_ptr, you can reimplement the program from Listing 58-5. The old program used the artifact map to manage the lifetime of all artifacts. Although convenient, there is no reason to tie artifacts to this map, because the map is used only for parsing. In a real program, most of its work lies in the actual building of targets, not parsing the input. All the parsing objects should be freed and long gone by the time the program is building targets.

Rewrite the artifact-lookup portion of Listing 58-5 to allocate artifact objects dynamically, using shared_ptr throughout to refer to artifact pointers. See Listing 63-2 for my solution.

Listing 63-2. Using Smart Pointers to Manage Artifacts

```
std::map<std::string, std::shared_ptr<artifact>> artifacts;

std::shared_ptr<artifact>
lookup_artifact(std::string const& name)
{
  std::shared_ptr<artifact> a{artifacts[name]};
  if (a.get() == nullptr)
  {
    a = std::make_shared<artifact>(name);
    artifacts[name] = a;
  }
  return a;
}
```

With a little more care, you could use unique_ptr instead of shared_ptr, but that results in greater changes to the rest of the code. You should always prefer unique_ptr to shared_ptr, due to the overhead of maintaining the reference count. But if you require shared ownership, shared_ptr is your choice. In all cases, there is no reason to use raw pointers instead of a smart pointer.

Smart Arrays

Recall from Exploration 61 that allocating a single object is completely different from allocating an array of objects. Thus, smart pointers must also distinguish between a smart pointer to a single object and a smart pointer to an array of objects. In the C++ standard, the distinction is well-defined: unique_ptr has separate specializations for scalars and arrays. On the other hand, shared_ptr works only with single objects by default. To work with arrays, you have to provide a second argument to the constructor: std::default_delete<T[]>(). The second argument tells the shared_ptr how to delete its pointer. The standard library provides std::default_delete<T[]> to delete a pointer using delete[]. For example:

```
std::shared_ptr<int> array_ptr{ new int[10], std::default_delete<int[]>{} };
```

Pimpls

No, that's not a spelling error. Although programmers have spoken for years about pimples and warts in their programs, often referring to unsightly but unavoidable bits of code, Herb Sutter associated the phrase *pointer-to-implementation* with these pimples to come up with the *pimpl* idiom.

In short, a pimpl is a class that hides implementation details in an implementation class, and the public interface object holds only a pointer to that implementation object. Instead of forcing the user of your class to allocate and de-allocate objects, manage pointers, and keep track of object lifetimes, you can expose a class that is easier to use. Specifically, the user can treat instances of the class as values, in the manner of int and other built-in types.

The pimpl wrapper manages the lifetime of the pimpl object. It typically implements the special member functions: copy and move constructors, copy and move assignment operators, and destructor. It delegates most of its other member functions to the pimpl object. The user of the wrapper never has to be concerned with any of this.

Thus, we will rewrite the artifact class so that it wraps a pimpl—that is, a pointer to an artifact_impl class. The artifact_impl class will do the real work, and artifact will merely forward all functions through its pimpl. The language feature that makes pimpls possible is declaring a class name without providing a definition of the class, as illustrated by the following:

```
class artifact_impl;
```

This class declaration, often called a *forward declaration*, informs the compiler that artifact_impl is the name of a class. The declaration doesn't provide the compiler with anything more about the class, so the class type is *incomplete*. You face a number of restrictions on what you can do with an incomplete type. In particular, you cannot define any objects or data members of that type, nor can you use an incomplete class as a function parameter or return type. You cannot refer to any members of an incomplete class. But you can use pointers or references to the type when you define objects, data members, function parameters, and return types. In particular, you can use a pointer to artifact_impl in the artifact class.

A normal class definition is a *complete* type definition. You can mix forward declarations with a class definition of the same class name. A common pattern is for a header, such as artifact.hpp, to declare a forward declaration; a source file then fills in the complete class definition.

The definition of the artifact class, therefore, can have a data member that is a pointer to the artifact_impl class, or even a smart pointer to artifact_impl, even though the compiler knows only that artifact_impl is a class but doesn't know any details about it. This means the artifact.hpp header file is independent of the implementation of artifact_impl. The implementation details are tucked away in a separate file, and the rest of your program can make use of the artifact class completely insulated from artifact_impl. In large projects, this kind of barrier is tremendously important.

Writing the artifact.hpp header is not difficult. Start with a forward declaration of artifact_impl. In the artifact class, the declarations of the member functions are the same as in the original class. Change the data members to a single pointer to artifact_impl. Finally, overload operator< for two artifact objects. Implement the comparison by comparing names. Read Listing 63-3 to see one possible implementation of this class.

Listing 63-3. Defining an `artifact` Pimpl Wrapper Class

```
#ifndef ARTIFACT_HPP_
#define ARTIFACT_HPP_

#include <chrono>
#include <memory>
#include <string>

class artifact_impl;
class artifact
{
public:
  typedef std::chrono::system_clock clock;
  artifact();
  artifact(std::string const& name);
  artifact(artifact const&) = default;
  ~artifact() = default;
  artifact& operator=(artifact const&) = default;

  std::string const& name()     const;
  clock::time_point  mod_time() const;
  std::string        expand(std::string str) const;

  void build();
  clock::time_point get_mod_time();

  void store_variable(std::string const& name, std::string const& value);

private:
  std::shared_ptr<artifact_impl> pimpl_;
};

inline bool operator<(artifact const& a, artifact const& b)
{
  return a.name() < b.name();
}

#endif // ARTIFACT_HPP_
```

The header defines the artifact class without any mention of artifact_impl, except for the pimpl_ data member.

The next step is to write the source file, artifact.cpp. This is where the compiler needs the full definition of the artifact_impl class, thus making artifact_impl a *complete class*, so include the artifact_impl.hpp header. The artifact class doesn't do much on its own. Instead, it just delegates every action to the artifact_impl class. See the details in Listing 63-4.

Listing 63-4. Implementing the `artifact` Class

```
#include "artifact.hpp"
#include "artifact_impl.hpp"

artifact::artifact() : pimpl_{std::make_shared<artifact_impl>()} {}

artifact::artifact(std::string const& name)
: pimpl_(std::make_shared<artifact_impl>(name))
{}

std::string const& artifact::name()
const
{
   return pimpl_->name();
}
```

```
artifact::clock::time_point artifact::mod_time()
const
{
   return pimpl_->mod_time();
}

std::string artifact::expand(std::string str)
const
{
   return pimpl_->expand(str);
}

void artifact::build()
{
   pimpl_->build();
}

artifact::clock::time_point artifact::get_mod_time()
{
   return pimpl_->get_mod_time();
}

void artifact::store_variable(std::string const& name, std::string const& value)
{
    pimpl_->store_variable(name, value);
}
```

You define the artifact_impl class in the artifact_impl.hpp header. This class looks nearly identical to the original artifact class. Listing 63-5 shows the artifact_impl class definition.

Listing 63-5. Defining the Artifact Implementation Class

```
#ifndef ARTIFACT_IMPL_HPP_
#define ARTIFACT_IMPL_HPP_

#include <cstdlib>
#include <chrono>
#include <memory>
#include <string>
#include "variables.hpp"

class artifact_impl
{
public:
  typedef std::chrono::system_clock clock;
  artifact_impl();
  artifact_impl(std::string const& name);
  artifact_impl(artifact_impl&&) = default;
  artifact_impl(artifact_impl const&) = delete;
  ~artifact_impl() = default;
  artifact_impl& operator=(artifact_impl&&) = default;
  artifact_impl& operator=(artifact_impl&) = delete;

  std::string const& name()     const { return name_; }
  clock::time_point  mod_time() const { return mod_time_; }

  std::string        expand(std::string str) const;
  void               build();
  clock::time_point  get_mod_time();
  void store_variable(std::string const& name, std::string const& value);
private:
  std::string name_;
  clock::time_point mod_time_;
  std::unique_ptr<variable_map> variables_;
};

#endif // ARTIFACT_IMPL_HPP_
```

The artifact_impl class is unsurprising. The implementation is just like the old artifact implementation, except the variables_ data member is now managed by unique_ptr instead of explicit code. That means the compiler writes the move constructor, move assignment operator, and destructor for you.

Now it's time to rewrite the lookup_artifact function yet again. **Rewrite Listings 59-4, 59-8, and 63-4 to use the new artifact class.** This time, the artifacts map stores artifact objects directly. The dependency_graph class will also have to use artifact instead of artifact*. See Listing 63-6 for one way to rewrite the program.

Listing 63-6. Rewriting the Program to Use the New `artifact` Value Class

```
#include <chrono>
#include <iostream>
#include <sstream>
#include <string>

#include "artifact.hpp"
#include "depgraph.hpp"  // Listing 58-5

#include "variables.hpp" // Listing 59-6


void parse_graph(std::istream& in, dependency_graph& graph)
{
  std::map<std::string, artifact> artifacts{};
  std::string line{};
  while (std::getline(in, line))
  {
    std::string target_name{}, dependency_name{};
    std::istringstream stream{line};
    if (stream >> target_name >> dependency_name)
    {
      artifact target{artifacts[expand(target_name, 0)]};
      std::string::size_type equal{dependency_name.find('=')};
      if (equal == std::string::npos)
      {
        // It's a dependency specification
        artifact dependency{artifacts[target.expand(dependency_name)]};
        graph.store_dependency(target, dependency);
      }
      else
        // It's a target-specific variable
        target.store_variable(dependency_name.substr(0, equal-1),
                              dependency_name.substr(equal+1));
    }
    else if (not target_name.empty())
    {
      std::string::size_type equal{target_name.find('=')};
      if (equal == std::string::npos)
        // Input line has a target with no dependency,
        // so report an error.
        std::cerr << "malformed input: target, " << target_name <<
                     ", must be followed by a dependency name\n";
      else
        global_variables[target_name.substr(0, equal)] =
                                          target_name.substr(equal+1);
    }
    // else ignore blank lines
  }
}

int main()
{
  dependency_graph graph{};

  parse_graph(std::cin, graph);

  try {
    // Get the sorted artifacts in reverse order.
    std::vector<artifact> sorted{};
    graph.sort(std::back_inserter(sorted));

    // Then print the artifacts in the correct order.
    for (auto it(sorted.rbegin()), end(sorted.rend());
         it != end;
         ++it)
    {
      std::cout << it->name() << '\n';
    }
  } catch (std::runtime_error const& ex) {
    std::cerr << ex.what() << '\n';
    return EXIT_FAILURE;
  }
}
```

As you can see, the code that uses artifact objects is simpler and easier to read. The complexity of managing pointers is pushed down into the artifact and artifact_impl classes. In this manner, the complexity is kept contained in one place and not spread throughout the application. Because the code that uses artifact is now simpler, it is less likely to contain errors. Because the complexity is localized, it is easier to review and test thoroughly. The cost is a little more development time, to write two classes instead of one, and a little more maintenance effort, because anytime a new function is needed in the artifact public interface, that function must also be added to artifact_impl. In many, many situations, the benefits far outweigh the costs, which is why this idiom is so popular.

The new artifact class is easy to use, because you can use it the same way you use an int. That is, you can copy it, assign it, store it in a container, etc., without concern about the size of an artifact object or the cost of copying it. Instead of treating an artifact as a big, fat object, or as a dangerous pointer, you can treat it as a value. Defining a class with *value semantics* makes it easy to use. Although it was more work to implement, the value artifact is the easiest incarnation to use for writing the application.

Iterators

Perhaps you've noticed the similarity between iterator syntax and pointer syntax. The C++ committee deliberately designed iterators to mimic pointers. Indeed, a pointer meets all the requirements of a random-access iterator, so you can use all the standard algorithms with a C-style array, as follows:

```
int data[4];
std::fill(data, data + 4, 42);
```

Thus, iterators are a form of smart pointer. Iterators are especially smart, because they come in five distinct flavors (see Exploration 44 for a reminder). Random-access iterators are just like pointers; other kinds of iterators have less functionality, so they are smart by being dumb.

Iterators can be just as dangerous as pointers. In their pure form, iterators are nearly as unchecked, wild, and raw as pointers. After all, iterators do not prevent you from advancing too far, from dereferencing an uninitialized iterator, from comparing iterators that point to different containers, etc. The list of unsafe practices with iterators is quite extensive.

Because these errors result in undefined behavior, a library implementer is free to choose any result for each kind of error. In the interest of performance, most libraries do not implement additional safety checks and push that back on the programmer, who can decide on his or her preference for a safety/performance trade-off.

If the programmer prefers safety to performance, some library implementations offer a debugging version that implements a number of safety checks. The debugging version of the standard library can check that iterators refer to the same container when comparing the iterators and throw an exception if they do not. An iterator is allowed to check that it is valid before honoring the dereference (*) operator. An iterator can ensure that it does not advance past the end of a container.

Thus, iterators are smart pointers, because they can be really, really smart. I highly recommend that you take full advantage of all safety features that your standard library offers. Remove checks one by one only after you have measured the performance of your program and found that one particular check degrades performance significantly, and you have the reviews and tests in place to give you confidence in the less safe code.

This completes your tour of pointers and memory. The next topic gets down into the bits and bytes of C++.

EXPLORATION 64

Working with Bits

This Exploration begins a series of Explorations that cover more advanced topics in the C++ type system. The series kicks off with an examination of how to work with individual bits. This Exploration begins with operators that manipulate integers at the bit level, then introduces bitfields—a completely different way of working with bits. The final topic is the bitset class template, which lets you work with bitsets of any size.

Integer As a Set of Bits

A common idiom in computer programming is to treat an integer as a bitmask. The bits can represent a set of small integers, such that a value *n* is a member of the set if the bit at position *n* is one; *n* is not in the set if the corresponding bit is zero. An empty set has the numeric value zero, because all bits are zero. To better understand how this works, consider the I/O stream formatting flags (introduced in Exploration 39).

Typically, you use manipulators to set and clear flags. For example, Exploration 17 introduced the skipws and noskipws manipulators. These manipulators set and clear the std::ios_base::skipws flag by calling the setf and unsetf member functions. In other words, the following statement:

```
std::cin >> std::noskipws >> read >> std::skipws;
```

is exactly equivalent to

```
std::cin.unsetf(std::ios_base::skipws);
std::cin >> read;
std::cin.setf(std::ios_base::skipws);
```

Other formatting flags include boolalpha (introduced in Exploration 12), showbase (Exploration 54), showpoint (display a decimal point even when it would otherwise be suppressed), and showpos (show a plus sign for positive numbers). Consult a C++ reference to learn about the remaining formatting flags.

A simple implementation of the formatting flags is to store the flags in an int and assign a specific bit position to each flag. A common way to write flags that you define in this manner is to use hexadecimal notation, as shown in Listing 64-1. Write a hexadecimal integer literal with 0x or 0X, followed by the base 16 value. Letters A through F in upper- or lowercase represent 10 through 15. (The C++ standard does not mandate any particular implementation of the formatting flags. Your library probably implements the formatting flags differently.)

Listing 64-1. An Initial Definition of Formatting Flags

```
typedef int fmtflags;
fmtflags const showbase  = 0x01;
fmtflags const boolalpha = 0x02;
fmtflags const skipws    = 0x04;
fmtflags const showpoint = 0x08;
fmtflags const showpos   = 0x10;
// etc. for other flags...
```

The next step is to write the setf and unsetf functions. The former function sets specific bits in a flags_ data member (of the std::ios_base class), and the latter clears bits. To set and clear bits, C++ provides some operators that manipulate individual bits in an integer. Collectively, they are called the *bitwise* operators.

The bitwise operators perform the usual arithmetic promotions and conversions (Exploration 25). The operators then perform their operation on successive bits in their arguments. The & operator implements bitwise *and*; the | operator implements bitwise inclusive *or*; and the ~ operator is a unary operator to perform bitwise complement. Figure 64-1 illustrates the bitwise nature of these operators (using & as an example).

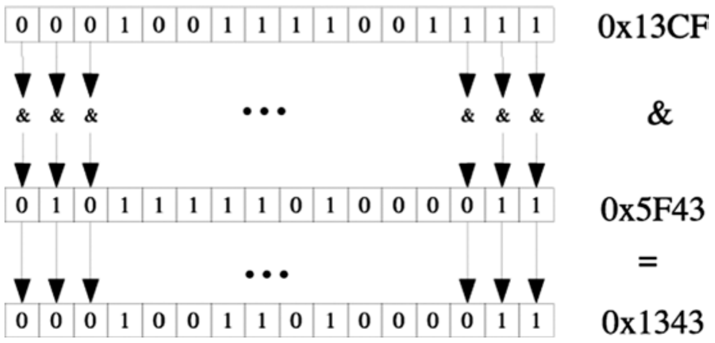


Figure 64-1. *How the & (bitwise and) operator works*

OPERATOR ABUSE

It may seem strange to you (it certainly does to me) that C++ uses the same operator to obtain the address of an object and to perform bitwise *and*. There are only so many characters to go around. The logical *and* operator is used for rvalue references. The asterisk does double duty for multiplication and dereferencing a pointer or iterator. The difference is whether the operator is unary (one operand) or binary (two operands). So there is no ambiguity, just unusual syntax. Later in this Exploration, you will learn that the I/O operators are repurposed shift operators. But don't worry, you will get used to it. Eventually.

Implement the setf function. This function takes a single fmtflags argument and sets the specified flags in the flags_ data member. Listing 64-2 shows a simple solution.

Listing 64-2. A Simple Implementation of the `setf` Member Function

```
void setf(fmtflags f)
{
   flags_ = flags_ | f;
}
```

The unsetf function is slightly more complicated. It must clear flags, which means setting the corresponding bits to zero. In other words, the argument specifies a bitmask in which each 1 bit means to clear (set to 0) the bit in flags_. **Write the unsetf function**. Compare your solution with Listing 64-3.

Listing 64-3. A Simple Implementation of the `unsetf` Member Function

```
void unsetf(fmtflags f)
{
   flags_ = flags_ & ~f;
}
```

Recall from Exploration 46 that various assignment operators combine an arithmetic operator with assignment. Assignment operators also exist for the bitwise functions, so you can write these functions even more succinctly, as shown in Listing 64-4.

Listing 64-4. Using Assignment Operators in the Flags Functions

```
void setf(fmtflags f)
{
   flags_ |= f;
}

void unsetf(fmtflags f)
{
   flags_ &= ~f;
}
```

Recall from Exploration 56 that the | operator combines I/O mode flags. Now you know that the flags are bits, and the I/O mode is a bitmask. Should the need arise, you can use any of the bitwise operators on I/O modes.

Bitmasks

Not all the flags are individual bits. The alignment flags, for example, can be left, right, or internal. The floating-point style can be fixed, scientific, hexfloat, or general. To represent three or four values, you need two bits. For these situations, C++ has a two-argument form of the setf function. The first argument specifies a mask of bits to set within the field, and the second argument specifies a mask of which bits to affect.

Using the same bitwise operators, you can define adjustfield as a two-bit-wide bitmask, for example, 0x300. If both bits are clear, that could mean left-adjustment; one bit set means right-adjustment; the other bit could mean "internal" alignment (align after a sign or 0x in a hexadecimal value). That leaves one more possible value (both bits set), but the standard library defines only three different alignment values.

Listing 64-5 hows one possible implementation of the adjustfield and floatfield masks and their associated values.

Listing 64-5. Declarations for Formatting Fields

```
fmtflags static constexpr adjustfield = 0x300;
fmtflags static constexpr left        = 0x000;
fmtflags static constexpr right       = 0x100;
fmtflags static constexpr internal    = 0x200;
fmtflags static constexpr floatfield  = 0xC00;
fmtflags static constexpr scientific  = 0x400;
fmtflags static constexpr fixed       = 0x800;
fmtflags static constexpr hexfloat    = 0xC00;
// general does not have a name; its value is zero
```

Thus, to set the alignment to right, one calls setf(right, adjustfield). **Write the two-argument form of the setf function**. Compare your solution with Listing 64-6.

Listing 64-6. Two-Argument Form of the `setf` Function

```
void setf(fmtflags flags_to_set, fmtflags field)
{
   flags_ &= ~field;
   flags_ |= flags_to_set;
}
```

One difficulty with defining bitfields in this fashion is that the numeric values can be hard to read, unless you've spent a lot of time working with hexadecimal values. Another solution is to use more familiar integers for all flags and fields and let the computer do the hard work by shifting those values into the correct positions.

Shifting Bits

Listing 64-7 shows another way to define the formatting fields. They represent the exact same values as shown in Listing 64-1, but they are a little easier to proofread.

Listing 64-7. Using Shift Operators to Define the Formatting Fields

```
int static constexpr boolalpha_pos = 0;
int static constexpr showbase_pos  = 1;
int static constexpr showpoint_pos = 2;
int static constexpr showpos_pos   = 3;
int static constexpr skipws_pos    = 4;
int static constexpr adjust_pos    = 5;
int static constexpr adjust_size   = 2;
int static constexpr float_pos     = 7;
int static constexpr float_size    = 2;

fmtflags static constexpr boolalpha   = 1 << boolalpha_pos;
fmtflags static constexpr showbase    = 1 << showbase_pos;
fmtflags static constexpr showpos     = 1 << showpos_pos;
fmtflags static constexpr showpoint   = 1 << showpoint_pos;
fmtflags static constexpr skipws      = 1 << showpoint_pos;
fmtflags static constexpr adjustfield = 3 << adjust_pos;
fmtflags static constexpr floatfield  = 3 << float_pos;

fmtflags static constexpr left     = 0 << adjust_pos;
fmtflags static constexpr right    = 1 << adjust_pos;
fmtflags static constexpr internal = 2 << adjust_pos;

fmtflags static constexpr fixed      = 1 << float_pos;
fmtflags static constexpr scientific = 2 << float_pos;
fmtflags static constexpr hexfloat   = 3 << float_pos;
```

The << operator (which looks just like the output operator) is the left-shift operator. It shifts its left-hand operator (which must be an integer) by the number of bit positions specified by the right-hand operator (also an integer). Vacated bits are filled with zero.

```
1 << 2  == 4
10 << 3 == 80
```

Although this style is more verbose, you can clearly see that the bits are defined with adjacent values. You can also easily see the size of multi-bit masks. If you have to add a new flag, you can do so without the need to recompute any other fields or flags.

What is the C++ right-shift operator? ________________. That's right: >>, which is also the input operator.

If the right-hand operand is negative, that reverses the direction of the shift. That is, a left shift by a negative amount is the same as right-shifting by a positive amount and vice versa. You can use the shift operators on integers but not on floating-point numbers. The right-hand operand cannot be greater than the number of bits in the left-hand operand. (Use the numeric_limits class template, introduced in Exploration 25, to determine the number of bits in a type, such as int.)

The C++ standard library overloads the shift operators for the I/O stream classes to implement the I/O operators. Thus, the >> and << operators were designed for shifting bits in an integer and were later usurped for I/O. As a result, the operator precedence is not quite right for I/O. In particular, the shift operators have a higher precedence than the bitwise operators, because that makes the most sense for manipulating bits. As a consequence, if, for instance, you want to print the result of a bitwise operation, you must enclose the expression in parentheses.

```
std::cout << "5 & 3 = " << (5 & 3) << '\n';
```

One caution when using the right-shift operator: The value of the bits that are filled in is implementation-defined. This can be particularly problematic with negative numbers. The value -1 >> 1 may be positive on some implementations and negative on others. Fortunately, C++ has a way to avoid this uncertainty, as the next section explains.

Safe Shifting with Unsigned Types

Every primitive integer type has a corresponding type that you declare with the unsigned keyword. These types are known—not surprisingly—as *unsigned* types. One key difference between ordinary (or signed) integer types and their unsigned equivalents is that unsigned types always shift in a zero when right-shifting. For this reason, unsigned types are preferable to signed types for implementing bitmasks.

```
typedef unsigned int fmtflags;
```

Write a program to determine how your C++ environment right-shifts negative values. Compare this with shifting unsigned values. Your program will certainly look different from mine, which is shown in Listing 64-8, but you should be able to recognize the key similarities.

Listing 64-8. Exploring How Negative and Unsigned Values Are Shifted

```
#include <iostream>
#include <string>

template<class T>
void print(std::string const& label, T value)
{
   std::cout << label << " = ";
   std::cout << std::dec << value << " = ";
   std::cout.width(8);
   std::cout.fill('0');
   std::cout << std::hex << std::internal << std::showbase << value << '\n';
}

int main()
{
   int i{~0}; // all bits set to 1; on most systems, ~0 == -1
   unsigned int u{~0u}; // all bits set to 1
   print("int >> 15", i >> 15);
   print("unsigned >> 15", u >> 15);
}
```

On my Linux x86 system, I see the following output:

```
int >> 15 = -1 = 0xffffffff
unsigned >> 15 = 131071 = 0x01ffff
```

which means right-shifting a signed value fills in the vacated bits with copies of the sign bit (a process known as *sign extensio*n, and that right-shifting an unsigned value works correctly by shifting in zero bits.

Signed and Unsigned Types

The plain int type is shorthand for signed int. That is, the int type has two sign flavors: signed int and unsigned int, the default being signed int. Similarly, short int is the same as signed short int, and long int is the same as signed long int. Thus, you have no reason to use the signed keyword with the integer types.

Like too many rules, however, this one has an exception: signed char. The char type comes in three flavors, not two: char, signed char, and unsigned char. All three types occupy the same amount of space (one byte). The plain char type has the same representation as either signed char or unsigned char, but it remains a distinct type. The choice is left to the compiler; consult your compiler's documentation to learn the equivalent char type for your implementation. Thus, the signed keyword has a use for the signed char type; the most common use for signed char is to represent a tiny, signed integer when conserving memory is important. Use plain char for text, signed char for tiny integers, and unsigned char for tiny bitmasks.

Unfortunately, the I/O stream classes treat signed char and unsigned char as text, not tiny integers or bitmasks. Thus, reading and writing tiny integers is harder than it should be, as demonstrated in the following:

```
signed char real_value{-42};
std::cout << static_cast<int>(real_value) << '\n';

int tmp{};
std::cin >> tmp;
if (tmp >= std::numeric_limits<signed char>::min() and
    tmp <= std::numeric_limits<signed char>::max())
{
  real_value = static_cast<signed char>(tmp);
  // do something with real_value
}
else // handle the error
```

Unsigned Literals

If an integer literal does not fit in a signed int, the compiler tries to make it fit into an unsigned int. If that works, the literal's type is unsigned int. If the value is too big for unsigned int, the compiler tries long, and then unsigned long, then long long, and finally unsigned long long, before giving up and issuing an error message.

You can force an integer to be unsigned with the u or U suffix. The U and L suffixes can appear in any order for an unsigned long literal. Use ULL for unsigned long long. (Remember that C++ permits lowercase l, but I recommend uppercase L to avoid confusion with numeral 1.)

```
1234u
4321UL
0xFFFFLu
```

One consequence of this flexibility is that you can't always know the type of an integer literal. For example, the type of 0xFFFFFFFF might be int on a 64-bit system. On some 32-bit systems, the type might be unsigned int, and on others, it might be unsigned long. The moral is to make sure you write code that works correctly, regardless of the precise type of an integer literal, which isn't difficult. For example, all the programs and fragments in this book work on any C++ compiler, regardless of the size of an int.

Type Conversions

A signed type and its unsigned counterpart always occupy the same amount of space. You can use static_cast (Exploration 25) to convert one to the other, or you can let the compiler implicitly perform the conversion, which can result in surprises, if you aren't careful. Consider the example in Listing 64-9.

Listing 64-9. Mixing Signed and Unsigned Integers

```
void show(unsigned u)
{
   std::cout << u << '\n';
}

int main()
{
   int i{-1};
   std::cout << i << '\n';
   show(i);
}
```

This results in the following output on my system:

```
-1
4294967295
```

If you mix signed and unsigned values in an expression (usually a bad idea), the compiler converts the signed value to unsigned, which often results in more surprises. This kind of surprise often arises in comparisons. Most compilers will at least warn you about the problem.

Listing 64-10. Mystery Program

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

template<class T>
void append(std::vector<T>& data, const T& value, int max_size)
{
  if (data.size() < max_size - 1)
    data.push_back(value);
}

int main()
{
  std::vector<int> data{};
  append(data, 10, 3);
  append(data, 20, 2);
  append(data, 30, 1);
  append(data, 40, 0);
  append(data, 50, 0);
  std::copy(data.begin(), data.end(),
          std::ostream_iterator<int>(std::cout, " "));
  std::cout << '\n';
}
```

Before you run the program, **predict what Listing 64-10 will print**.

____

____

____

Try it. Were you correct? ________________ **Explain what the program does**.

____

____

____

The program succeeds in appending 10 to data because the vector size is zero, which is less than 2. The next call to append, however, does nothing, because the vector size is 1, and max_size - 1 is also 1. The next call fails for a similar reason. So why does the next call succeed in appending 40 to data? Because max_size is 0, you might think the comparison would be with -1, but -1 is signed, and data.size() is unsigned. Therefore, the compiler converts -1 to unsigned, which is an implementation-defined conversion. On typical workstations, -1 converts to the largest unsigned integer, so the test succeeds.

The first moral of the story is to avoid expressions that mix signed and unsigned values. Your compiler might help you here by issuing warnings when you mix signed and unsigned values. A common source for unsigned values is from the size() member functions in the standard library, which all return an unsigned result. You can reduce the chances for surprises by using one of the standard typedefs for sizes, such as std::size_t (defined in <cstdlib> and <cstddef>), which is an implementation-defined unsigned integer type. The standard containers all define a member type, size_type, to represent sizes and similar values for that container. Use these typedefs for your variables when you know you have to store sizes, indices, or counts.

"That's easy!" you say. "Just change the declaration of max_size to std::vector<T>::size_type, and problem solved!" Maybe you can avoid this kind of problem by sticking with the standard member typedefs, such as size_type and difference_type (Exploration 43). Take a gander at Listing 64-11 and see what you think.

Listing 64-11. Another Mystery Program

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

/** Return the index of a value in a range.
 * Look for the first occurrence of @p value in the range
 * [<tt>first</tt>, <tt>last</tt>), and return the zero-based
 * index or -1 if @p value is not found.
 * @param first The start of the range to search
 * @param last One past the end of the range to search
 * @param value The value to search for
 * @return [0, size), such that size == last-first, or -1
 */
template<class InputIter>
typename std::iterator_traits<InputIter>::difference_type
index_of(InputIter first, InputIter last,
         typename std::iterator_traits<InputIter>::value_type const& value)
{
   InputIter iter{std::find(first, last, value)};
   if (iter == last)
      return -1;
   else
      return std::distance(first, iter);
}

/** Determine whether the first occurrence of a value in a container is
 * in the last position in the container.
 * @param container Any standard container
 * @param value The value to search for.
 * @return true if @p value is at the last position,
 *         or false if @p value is not found or at any other position.
 */
template<class T>
bool is_last(T const& container, typename T::value_type const& value)
{
    return index_of(container.begin(), container.end(), value) ==
           container.size() - 1;
}
```

```
int main()
{
   std::vector<int> data{};
   if (is_last(data, 10))
      std::cout << "10 is the last item in data\n";
}
```

Predict the output before you run the program in Listing 64-11.

____

Try it. What do you actually get?

____

I get "10 is the last item in data," even though data is clearly empty. Can you spot the conceptual error that I committed? In a standard container, the difference_type typedef is always a signed integral type. Thus, index_of() always returns a signed value. I made the mistake of thinking that the signed value -1 would always be less than any unsigned value because they are always 0 or more. Thus, is_last() would not have to check for an empty container as a special case.

What I failed to take into account is that when a C++ expression mixes signed and unsigned values, the compiler converts the signed value to unsigned. Thus, the signed result from index_of becomes unsigned, and -1 becomes the largest possible unsigned value (on a typical two's complement system, such as most desktop computers). If the container is empty, size() is zero, and size() - 1 (which the compiler interprets as size() - 1u) is also the largest possible unsigned integer.

If you are fortunate, your compiler issues a warning about comparing signed and unsigned values. That gives you a hint that something is wrong. **Fix the program. Compare your solution with Listing 64-12**.

Listing 64-12. Fixing the Second Mystery Program

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

/** Return the index of a value in a range.
 * Look for the first occurrence of @p value in the range
 * [<tt>first</tt>, <tt>last</tt>), and return the zero-based
 * index or -1 if @p value is not found.
 * @param first The start of the range to search
 * @param last One past the end of the range to search
 * @param value The value to search for
 * @return [0, size), such that size == last-first, or -1
 */
template<class InputIter>
typename std::iterator_traits<InputIter>::difference_type
index_of(InputIter first, InputIter last,
         typename std::iterator_traits<InputIter>::value_type const& value)
{
   InputIter iter{std::find(first, last, value)};
   if (iter == last)
      return -1;
   else
      return std::distance(first, iter);
}
```

```
/** Determine whether the first occurrence of a value in a container is
 * in the last position in the container.
 * @param container Any standard container
 * @param value The value to search for.
 * @return true if @p value is at the last position,
 *    or false if @p value is not found or at any other position.
 */
template<class T>
bool is_last(T const& container, typename T::value_type const& value)
{
   typename T::difference_type
        pos(index_of(container.begin(), container.end(), value));
  auto last_index(static_cast<typename T::difference_type>(container.size() - 1));
   return pos != -1 and pos == last_index;
}

int main()
{
   std::vector<int> data{};
   if (is_last(data, 10))
      std::cout << "10 is the last item in data\n";
}
```

The second moral of the story is not to use unsigned types if you don't have to. Most of the time, signed types work just as well. Just because a type's range of legal values happens to be non-negative is not a reason to use an unsigned type. Doing so only complicates any code that must cooperate with the unsigned type.

■ **Tip** When using the standard library, make use of the typedefs and member typedefs that it provides. When you have control over the types, use signed types for all numeric types, including sizes, and reserve the unsigned types for bitmasks. And always be very, very careful every time you write an expression that uses an unsigned type with other integers.

Overflow

Until now, I've told you to ignore arithmetic overflow. That's because it's a difficult topic. Strictly speaking, if an expression involving signed integers or floating-point numbers overflows, the results are undefined. In reality, your typical desktop system wraps integer overflow (so adding two positive numbers can yield a negative result). Overflow of floating-point numbers can yield infinity, or the program may terminate.

If you must prevent overflow, you should check values before evaluating an expression, to ensure the expression will not overflow. Use std::numeric_limits<> to check the min() and max() values for the type.

If you explicitly cast a signed value to a type, such that the value overflows the destination type, the results are not so dire. Instead of undefined behavior, the results are defined by the implementation. Most implementations simply discard the excess bits. Therefore, for maximum safety and portability, you should check for overflow. Use numeric_limits (Exploration 25) to learn the maximum or minimum value of a type.

Unsigned integers are different. The standard explicitly permits unsigned arithmetic to overflow. The result is to discard any extra high-order bits. Mathematically speaking, this means unsigned arithmetic is modulo 2^n, where n is the number of bits in the unsigned type. If you have to perform arithmetic that you know may overflow, and you want the values to wrap around without reporting an error, use static_cast to cast to the corresponding unsigned types, perform the arithmetic, and static_cast back the original types. The static_casts have no impact on performance, but they clearly tell the compiler and the human what's going on.

Introducing Bitfields

A *bitfield* is a way to partition an integer within a class into individual bits or masks of adjacent bits. Declare a bitfield using an unsigned integer type or bool, the field name, a colon, and the number of bits in the field. Listing 64-13 shows how you might store the I/O formatting flags using bitfields.

Listing 64-13. Declaring Formatting Flags with Bitfields

```
struct fmtflags
{
   bool skipws_f :         1;
   bool boolalpha_f:       1;
   bool showpoint_f:       1;
   bool showbase_f:        1;
   bool showpos_f:         1;
   unsigned adjustfield_f: 2;
   unsigned floatfield_f:  2;

   static unsigned constexpr left     = 0;
   static unsigned constexpr right    = 1;
   static unsigned constexpr internal = 2;

   static unsigned constexpr fixed      = 1;
   static unsigned constexpr scientific = 2;
   static unsigned constexpr hexfloat   = 3;
};
```

Use a bitfield member the way you would use any other data member. For example, to set the skipws flag, use

```
flags.skipws_f = true;
```

and to clear the flag, use the following:

```
flags.skipws_f = false;
```

To select scientific notation, try the line that follows:

```
flags.floatfield_f = fmtflags::scientific;
```

As you can see, code that uses bitfields is easier to read and write than the equivalent code using shift and bitwise operators. That's what makes bitfields popular. On the other hand, it is hard to write functions such as setf and unsetf. It is hard to get or set multiple, nonadjacent bits at one time. That's why your library probably doesn't use bitfields to implement I/O formatting flags.

Another limitation is that you cannot take the address of a bitfield (with the & operator), because an individual bit is not directly addressable in the C++ memory model.

Nonetheless, the clarity that bitfields offer puts them at the top of the list when choosing an implementation. Sometimes, other factors knock them off the list, but you should always consider bitfields first. With bitfields, you don't have to be concerned with bitwise operators, shift operators, mixed-up operator precedence, and so on.

Portability

The C++ standard leaves several details up to each implementation. In particular, the order of bits in a field is left up to the implementation. A bitfield cannot cross a word boundary where the definition of a *word* is also left up to the implementation. Popular desktop and workstation computers often use 32 bits or 64 bits, but there is no guarantee that a word is the same size as an int. An unnamed bitfield of size zero tells the compiler to insert pad bits, so that the subsequent declaration aligns on a word boundary.

```
class demo {
  unsigned bit0 : 1;
  unsigned bit1 : 1;
  unsigned bit2 : 3;
  unsigned      : 0;
  unsigned word1: 2;
};
```

The size of a demo object depends on the implementation. Whether bit0 is the least or most significant bit of a demo's actual implementation also varies from one system to another. The number of pad bits between bit2 and word1 also depends on the implementation.

Most code does not have to know the layout of the bits in memory. On the other hand, if you are writing code that interprets the bits in a hardware control register, you have to know the order of bits, the exact nature of padding bits, and so on. But you probably aren't expecting to write highly portable code, anyway. In the most common case, when you are trying to express a compact set of individual set members or small bitmasks, bitfields are wonderful. They are easy to write and easy to read. They are limited, however, to a single word, often 32 bits. For larger bitfields, you must use a class, such as std::bitset.

The bitset Class Template

Sometimes you have to store more bits than can fit in an integer. In that case, you can use the std::bitset class template, which implements a fixed-size string of bits of any size.

The std::bitset class template takes one template argument: the number of bits in the set. Use a bitset object the way you would any other value. It supports all the bitwise and shift operators, plus a few member functions for further convenience. Another nifty trick that bitset can perform is the subscript operator, which lets you access individual bits in the set as discrete objects. The right-most (least significant) bit is at index zero. Construct a bitset from an unsigned long (to set the least-significant bits of the bitset, initializing the remaining bits to zero) or from a string of '0' and '1' characters, as illustrated in Listing 64-14.

Listing 64-14. Example of Using `std::bitset`

```
#include <bitset>
#include <cstdlib>
#include <iostream>
#include <string>

/** Find the first 1 bit in a bitset, starting from the most significant bit.
 * @param bitset The bitset to examine
 * @return A value in the range [0, bitset.size()-1) or
 *    size_t(-1) if bitset.none() is true.
 */
```

```
template<std::size_t N>
std::size_t first(std::bitset<N> const& bitset)
{
   for (std::size_t i{bitset.size()}; i-- != 0;)
      if (bitset.test(i))
         return i;
   return std::size_t(-1);
}

int main()
{
   std::bitset<50> lots_o_bits{std::string{"1011011101111011111011111101111111"}};
   std::cout << "bitset: " << lots_o_bits << '\n';
   std::cout << "first 1 bit: " << first(lots_o_bits) << '\n';
   std::cout << "count of 1 bits: " << lots_o_bits.count() << '\n';
   lots_o_bits[first(lots_o_bits)] = false;
   std::cout << "new first 1 bit: " << first(lots_o_bits) << '\n';
   lots_o_bits.flip();
   std::cout << "bitset: " << lots_o_bits << '\n';
   std::cout << "first 1 bit: " << first(lots_o_bits) << '\n';
}
```

In Exploration 25, I presented static_cast<> as a way to convert one integer to a different type. Listing 64-14 demonstrates another way to convert integer types, using constructor and initializer syntax: std::size_t(-1) or std::size{-1}. For a simple type conversion, this syntax is often easier to read than static_cast<>. I recommend using this syntax only when converting literals; use static_cast<> for more complicated expressions.

Unlike working with bitfields, most of the behavior of bitset is completely portable. Thus, every implementation gives the same results when running the program in Listing 64-14. The following output displays those results:

```
bitset: 00000000000000001011011101111011111011111101111111
first 1 bit: 33
count of 1 bits: 28
new first 1 bit: 31
bitset: 11111111111111111100100010000100000100000010000000
first 1 bit: 49
```

Write a function template, find_pair, that takes two arguments: a bitset to search, and a bool value to compare. The function searches for the first pair of adjacent bits that are equal to the second argument and returns the index of the most significant bit of the pair. **What should the function return if it cannot find a matching pair of bits? Write a simple test program too**.

Compare your solution with mine, which is presented in Listing 64-15.

Listing 64-15. The `find_pair` Function and Test Program

```
#include <bitset>
#include <cassert>
#include <cstdlib>
#include <iostream>
```

```
template<std::size_t N>
std::size_t find_pair(std::bitset<N> const& bitset, bool value)
{
   if (bitset.size() >= 2)
      for (std::size_t i{bitset.size()}; i-- != 1; )
         if (bitset[i] == value and bitset[i-1] == value)
            return i;
   return std::size_t(-1);
}

int main()
{
   std::size_t const not_found{~0u};
   std::bitset<0> bs0{};
   std::bitset<1> bs1{};
   std::bitset<2> bs2{};
   std::bitset<3> bs3{};
   std::bitset<100> bs100{};

   assert(find_pair(bs0, false) == not_found);
   assert(find_pair(bs0, true) == not_found);
   assert(find_pair(bs1, false) == not_found);
   assert(find_pair(bs1, true) == not_found);
   assert(find_pair(bs2, false) == 1);
   assert(find_pair(bs2, true) == not_found);
   bs2[0] = true;
   assert(find_pair(bs2, false) == not_found);
   assert(find_pair(bs2, true) == not_found);
   bs2.flip();
   assert(find_pair(bs2, false) == not_found);
   assert(find_pair(bs2, true) == not_found);
   bs2[0] = true;
   assert(find_pair(bs2, false) == not_found);
   assert(find_pair(bs2, true) == 1);
   assert(find_pair(bs3, false) == 2);
   assert(find_pair(bs3, true) == not_found);
   bs3[2].flip();
   assert(find_pair(bs3, false) == 1);
   assert(find_pair(bs3, true) == not_found);
   bs3[1].flip();
   assert(find_pair(bs3, false) == not_found);
   assert(find_pair(bs3, true) == 2);
   assert(find_pair(bs100, false) == 99);
   assert(find_pair(bs100, true) == not_found);
   bs100[50] = true;
   assert(find_pair(bs100, true) == not_found);
   bs100[51] = true;
   assert(find_pair(bs100, true) == 51);
}
```

Although bitset is not widely used, when you need it, it can be extremely helpful. The next Exploration covers a language feature that is much more widely used than bitset: enumerations.

EXPLORATION 65

Enumerations

The final mechanism for defining types in C++ is the enum keyword, which is short for *enumeration*. Enumerations in C++ 11 come in two flavors. One flavor originated in C and has some strange quirks. The other flavor addresses those quirks and will probably make more sense to you. This Exploration starts with the new flavor.

Scoped Enumerations

An enumerated type is a user-defined type that defines a set of identifiers as the values of the type. Define an enumerated type with the enum class keywords, followed by the name of your new type, followed by an optional colon and integer type, followed by the enumerated literals in curly braces. Feel free to substitute struct for class; they are interchangeable in an enum declaration. The following code shows some examples of enumerated types:

```
enum class color { black, red, green, yellow, blue, magenta, cyan, white };
enum class sign : char { negative, positive };
enum class flags : unsigned { boolalpha, showbase, showpoint, showpos, skipws };
enum class review : int { scathing = -2, negative, neutral, positive, rave };
```

A scoped enum definition defines a brand-new type that is distinct from all other types. The type name is also a scope name, and the names of all the enumerators are declared in that scope. Because enumerators are scoped, you can use the same enumerator name in multiple scoped enumerations.

The type after the colon must be an integral type. If you omit the colon and type, the compiler implicitly uses int. This type is called the *underlying type*. The enumerated value is stored as though it were a value of the underlying type.

Each enumerator names a compile-time constant. The type of each enumerator is the enumeration type. You can obtain the integral value by casting the enumerated type to its underlying type, and you can cast an integral type to the enumerated type. The compiler will *not* perform these conversions automatically. Listing 65-1 shows one way to implement the std::ios_base::openmode type (refresh your memory in Exploration 14). The type must support the bitwise operators to combine out, trunc, app, etc., which it can provide by converting enumerated value to unsigned, performing the operation, and casting back to openmode.

Listing 65-1. One Way to Implement openmode Type

```
enum class openmode : unsigned { in, out, binary, trunc, app, ate };

openmode operator|(openmode lhs, openmode rhs)
{
   return static_cast<openmode>(
     static_cast<unsigned>(lhs) | static_cast<unsigned>(rhs) );
}
```

```
openmode operator&(openmode lhs, openmode rhs)
{
   return static_cast<openmode>(
     static_cast<unsigned>(lhs) & static_cast<unsigned>(rhs) );
}

openmode operator~(openmode arg)
{
   return static_cast<openmode>( ~static_cast<unsigned>(arg) );
}
```

When you declare an enumerator, you can provide an integral value by following the enumerator name with an equal sign and a constant expression. The expression can use enumerators declared earlier in the same type as integral constants. If you omit a value, the compiler adds one to the previous value. If you omit a value for the first enumerator, the compiler uses zero. For example:

```
enum class color : unsigned { black, red=0xff0000, green=0x00ff00, blue=0x0000ff,
     cyan = blue|green, yellow=red|green, magenta=red|blue, white=red|blue|green };
```

You can forward declare an enumerated type, similar to the way you can forward declare a class type. If you omit the curly-braced list of enumerators, it tells the compiler the type name and its underlying type, so you can use the type to declare function parameters, data members, and so on. You can provide the enumerators in a separate declaration:

```
enum class deferred : short;
enum class deferred : short { example, of, forward, declared, enumeration };
```

A forward-declared enumeration is also called an *opaque declaration*. One way to use an opaque declaration is to declare the type in a header file, but provide the enumerators in a separate source file. If the opaque type is declared in a class or namespace, be sure to qualify the type's name when you provide the full type definition. For example, if the header contains

```
class demo {
  enum hidden : unsigned;
  ...
};
```

The source file might declare enumerators that are for use solely by the implementation, and not the user of the class as follows:

```
#include "demo.hpp"
enum demo::hidden : unsigned { a, b, c };
```

The compiler implicitly defines the comparison operators for enumerators. As you probably expect, they work by comparing the underlying integral values. The compiler provides no other operators, such as I/O, increment, decrement, etc.

Scoped enumerations pretty much work the way one would expect an enumerated type to work. Given the name "scoped" enumerations, you must surely be asking yourself, "What is an unscoped enumeration?" You are in for a surprise.

Unscoped Enumerations

An unscoped enumerated type defines a new type, distinct from other types. The new type is an integer bitmask type, with a set of predefined mask values. If you do not specify an underlying type, the compiler chooses one of the built-in integral types; the exact type is implementation-defined. Define an unscoped enumeration in the same manner as a scoped enumeration, except you must omit the class (or struct) keyword.

The compiler does not define the arithmetic operators for enumerated types, leaving you free to define these operators. The compiler implicitly converts an unscoped enumerated value to its underlying integer value, but to convert back, you must use an explicit type cast. Use the enumerated type name in the manner of a constructor with an integer argument or use static_cast.

```
enum color { black, blue, green, cyan, red, magenta, yellow, white };
int c1{yellow};
color c2{ color(c1 + 1) };
color c3{ static_cast<color>(c2 + 1) };
```

Calling an enum a "bitmask" may strike you as odd, but that is how the standard defines the implementation of an unscoped enumeration. Suppose you define the following enumeration:

```
enum sample { low=4, high=256 };
```

The permissible values for an object of type sample are all values in the range [sample(0), sample(511)]. The permissible values are all the bitmask values that fit into a bitfield that can hold the largest and smallest bitmask values among the enumerators. Thus, in order to store the value 256, the enumerated type must be able to store up to 9 bits. As a side effect, any 9-bit value is valid for the enumerated type, or integers up to 511.

You can use an enumerated type for a bitfield (Exploration 64). It is your responsibility to ensure that the bitfield is large enough to store all the possible enumeration values, as demonstrated in the following:

```
class demo {
  color okay  : 3; // big enough
  color small : 2; // oops, not enough bits, but valid C++
};
```

The compiler will let you declare a bitfield that is too small; if you are fortunate, your compiler will warn you about the problem.

C++ inherits this definition of enum from C. The earliest implementations lacked a formal model, so when the standards committee tried to nail down the formal semantics, the best they could do was to capture the behavior of extant compilers. In C++ 11, they invented scoped enumerators, to try to provide a rational way to define enumerations.

Strings and Enumerations

One deficiency of scoped and unscoped enumerations is I/O support. C++ does not implicitly create any I/O operators for enumerations. You are on your own. Unscoped enumerators can be implicitly promoted to the underlying type and printed as an integer, but that's all. If you want to use the names of the enumerators, you must implement the I/O operators yourself. One difficulty is that C++ gives you no way to discover the literal names with any reflection or introspection mechanism. Instead, you have to duplicate information and hope you don't make any mistakes.

In order to implement I/O operators that can read and write strings, you must be able to map strings to enumerated values and vice versa. Assume that conversions to and from strings will occur frequently. **What data structure should you use to convert strings to enumerated values?** ________________ **What data structure should you use for the reverse conversion?** ________________If you know that the enumerators are not arbitrary values but follow the implicit values that the compiler assigns, you can use std::array to map values to strings; otherwise, use std::unordered_map. To reduce the amount of redundant information in the program, write a helper function to populate both data structures simultaneously.

Suppose you define a language type that enumerates your favorite computer languages. **Implement the to_string(language) function to return a language as a string and a function, from_string(), to convert a string to a language. Throw std::out_of_range if the input is not valid. Write a function, void initialize_languages(), that initializes the internal data structures**.

See Listing 65-2 for the string conversion code for the language class.

Listing 65-2. Mapping a `language` Value to and from Strings

```
#include <array>
#include <istream>
#include <stdexcept>
#include <string>
#include <vector>
#include <unordered_map>

enum class language { apl, low=apl, c, cpp, forth, haskell, jovial, lisp, scala, high=scala };

std::unordered_map<std::string, language> from_string_map_;
std::array<std::string, 8> to_string_map_;

void store(std::string const& name, language lang)
{
   from_string_map_.emplace(name, lang);
   to_string_map_.at(static_cast<int>(lang)) = name;
}

void initialize_language()
{
   store("apl", language::apl);
   store("c", language::c);
   store("cpp", language::cpp);
   store("forth", language::forth);
   store("haskell", language::haskell);
   store("jovial", language::jovial);
   store("lisp", language::lisp);
   store("scala", language::scala);
}

std::string const& to_string(language lang)
{
   return  to_string_map_.at(static_cast<int>(lang));
}

language from_string(std::string const& text)
{
   return from_string_map_.at(text);
}
```

```
std::ostream& operator<<(std::ostream& stream, language lang)
{
   stream << to_string(lang);
   return stream;
}

std::istream& operator>>(std::istream& stream, language& lang)
{
   std::string text;
   if (stream >> text)
      lang = from_string(text);
   return stream;
}
```

The next task is to ensure that initialize_language() is called. The common idiom is to define an initialization class. Call it initializer. Its constructor calls initialize_language(). Thus, constructing a single instance of initializer ensures that the language class is initialized, as shown in Listing 65-3.

Listing 65-3. Automatically Initializing the Language Data Structures

```
class initializer {
public:
   initializer() { initialize_language(); }
};
initializer init;
```

Objects at namespace or global scope are constructed before main() begins or before the use of any function or object in the same file. Thus, the init object is constructed early, which calls initialize_language(), which in turn ensures the language data structures are properly initialized.

The one difficulty is when another global initializer has to use language. C++ offers no convenient way to ensure that objects in one file are initialized before objects in another file. Within a single file, objects are initialized in the order of declaration, starting at the top of the file. Dealing with this issue is beyond the scope of this book. None of the examples in this book depends on the order of initialization across files. In fact, most programs don't face this problem, because they don't use global or namespace scoped objects. So we can return to the immediate problem: reading and writing language values.

Revisiting Projects

Now that you know all about enumerations, consider how you could improve some previous projects. For example, in Exploration 35, we wrote a constructor for the point class that uses a bool to distinguish between Cartesian and polar coordinate systems. Because it is not obvious whether true means Cartesian or polar, a better solution is to use an enumerated type, such as the following:

```
enum class coordinate_system : bool { cartesian, polar };
```

Another example that can be improved with enumerations is the card class, from Listing 53-5. Instead of using int constants for the suits, use an enumeration. You can also use an enumeration for the rank. The enumeration has to specify enumerators: the number cards and ace, jack, queen, and king. Choose appropriate values so that you can cast an integer in the range [2, 10] to rank and get the desired value. You will have to implement operator++ for suit and rank. Write your new, improved card class and compare it with my solution in Listing 65-4.

Listing 65-4. Improving the card Class with Enumerations

```
#ifndef CARD_HPP_
#define CARD_HPP_

#include <istream>
#include <ostream>

enum class suit { nosuit, diamonds, clubs, hearts, spades };
enum class rank { norank=0, r2=2, r3, r4, r5, r6, r7, r8, r9, r10, jack, queen, king, ace };

suit& operator++(suit& s)
{
   if (s == suit::spades)
      s = suit::diamonds;
  else
     s = static_cast<suit>(static_cast<int>(s) + 1);
  return s;
}

rank operator++(rank& r)
{
   if (r == rank::norank or r == rank::ace)
      r = rank::r2;
   else
      r = static_cast<rank>(static_cast<int>(r) + 1);
   return r;
}

/// Represent a standard western playing card.
class card
{
public:
  card() : rank_(rank::norank), suit_(suit::nosuit) {}
  card(rank r, suit s) : rank_(r), suit_(s) {}

  void assign(rank r, suit s);
  suit get_suit() const { return suit_; }
  rank get_rank() const { return rank_; }
private:
  rank rank_;
  suit suit_;
};

bool operator==(card a, card b);
std::ostream& operator<<(std::ostream& out, card c);
std::istream& operator>>(std::istream& in, card& c);

/// In some games, Aces are high. In other Aces are low. Use different
/// comparison functors depending on the game.
bool acehigh_compare(card a, card b);
bool acelow_compare(card a, card b);
```

```
/// Generate successive playing cards, in a well-defined order,
/// namely, 2-10, J, Q, K, A. Diamonds first, then Clubs, Hearts, and Spades.
/// Roll-over and start at the beginning again after generating 52 cards.
class card_generator
{
public:
  card_generator();
  card operator()();
private:
  card card_;
};

#endif
```

What other projects can you improve with enumerations?

EXPLORATION 66

Multiple Inheritance

Unlike some other object-oriented languages, C++ lets a class have more than one base class. This feature is known as *multiple inheritance*. Several other languages permit a single base class and introduce a variety of mechanisms for pseudo-inheritance, such as Java interfaces and Ruby mix-ins and modules. Multiple inheritance in C++ is a superset of all these other behaviors.

Multiple Base Classes

Declare more than one base class by listing all the base classes in a comma-separated list. Each base class gets its own access specifier, as demonstrated in the following:

```
class derived : public base1, private base2, public base3 {
};
```

As with single inheritance, the derived class has access to all the non-private members of all of its base classes. The derived class constructor initializes all the base classes in order of declaration. If you have to pass arguments to any base class constructor, do so in the initializer list. As with data members, the order of initializers does not matter. Only the order of declaration matters, as illustrated in Listing 66-1.

Listing 66-1. Demonstrating the Order of Initialization of Base Classes

```
#include <iostream>
#include <ostream>
#include <string>

class visible {
public:
    visible(std::string&& msg) : msg_{std::move(msg)} { std::cout << msg_ << '\n'; }
    std::string const& msg() const { return msg_; }
private:
    std::string msg_;
};

class base1 : public visible {
public:
   base1(int x) : visible{"base1 constructed"}, value_{x} {}
   int value() const { return value_; }
private:
   int value_;
};

class base2 : public visible {
public:
   base2(std::string const& str) : visible{"base2{" + str + "} constructed"} {}
};

class base3 : public visible {
public:
   base3() : visible{"base3 constructed"} {}
   int value() const { return 42; }
};

class derived : public base1, public base2, public base3 {
public:
   derived(int i, std::string const& str) : base3{}, base2{str}, base1{i} {}
   int value() const { return base1::value() + base3::value(); }
   std::string msg() const
  {
     return base1::msg() + "\n" + base2::msg() + "\n" + base3::msg();
  }
};

int main()
{
   derived d{42, "example"};
}
```

Your compiler may issue a warning when you compile the program, pointing out that the order of base classes in derived's initializer list does not match the order in which the initializers are called. Running the program demonstrates that the order of the base classes controls the order of the constructors, as shown in the following output:

```
base1 constructed
base2{example} constructed
base3 constructed
```

Figure 66-1 illustrates the class hierarchy of Listing 66-1. Notice that each of the base1, base2, and base3 classes has its own copy of the visible base class. Don't be concerned now, but this point will arise later, so pay attention.

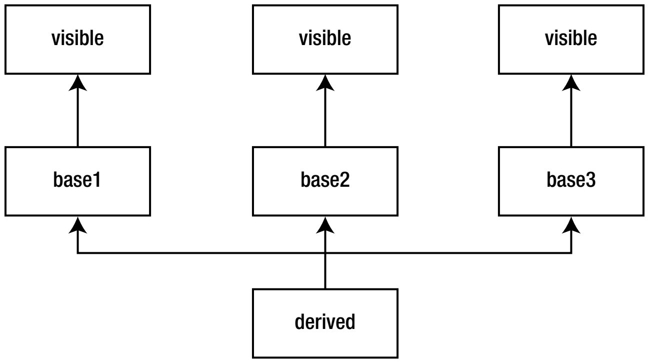


Figure 66-1. *UML diagram of classes in Listing 66-1*

If two or more base classes have a member with the same name, you must indicate to the compiler which of them you mean, if you want to access that particular member. Do this by qualifying the member name with the desired base class name when you access the member in the derived class. See the examples in the derived class in Listing 66-1. **Change the main() function to the following**:

```
int main()
{
   derived d{42, "example"};
   std::cout << d.value() << '\n' << d.msg() << '\n';
}
```

Predict the output from the new program.

____

____

____

____

____

____

____

Compare your results with the following output I got:

```
base1 constructed
base2{example} constructed
base3 constructed
84
base1 constructed
base2{example} constructed
base3 constructed
```

Virtual Base Classes

Sometimes you don't want a separate copy of a common base class. Instead, you want a single instance of the common base class, and every class shares that one common instance. To share base classes, insert the virtual keyword when declaring the base class. The virtual keyword can come before or after the access specifier; convention is to list it first.

■ **Note** C++ overloads certain keywords, such as static, virtual, and delete. Virtual base classes have no relationship with virtual functions. They just happen to use the same keyword.

Imagine changing the visible base class to be virtual when each of base1, base2, and base3 derive from it. **Can you think of any difficulty that might arise?**

____

Notice that each of the classes that inherit from visible pass a different value to the constructor for visible. If you want to share a single instance of visible, you have to pick one value and stick with it. To enforce this rule, the compiler ignores all the initializers for a virtual base class, except the one that it requires in the most-derived class (in this case, derived). Thus, to change visible to be virtual, not only must you change the declarations of base1, base2, and base3, but you must also change derived. When derived initializes visible, it initializes the sole, shared instance of visible. **Try it**. Your modified program should look something like Listing 66-2.

Listing 66-2. Changing the Inheritance of Visible to Virtual

```
#include <iostream>
#include <ostream>
#include <string>

class visible {
public:
    visible(std::string&& msg) : msg_{std::move(msg)} { std::cout << msg_ << '\n'; }
    std::string const& msg() const { return msg_; }
private:
    std::string msg_;
};

class base1 : virtual public visible {
public:
   base1(int x) : visible{"base1 constructed"}, value_{x} {}
   int value() const { return value_; }
private:
   int value_;
};

class base2 : virtual public visible {
public:
   base2(std::string const& str) : visible{"base2{" + str + "} constructed"} {}
};
```

```
class base3 : virtual public visible {
public:
   base3() : visible{"base3 constructed"} {}
   int value() const { return 42; }
};

class derived : public base1, public base2, public base3 {
public:
   derived(int i, std::string const& str)
   : base3{}, base2{str}, base1{i}, visible{"derived"}
   {}
   int value() const { return base1::value() + base3::value(); }
   std::string msg() const
   {
     return base1::msg() + "\n" + base2::msg() + "\n" + base3::msg();
   }
};

int main()
{
   derived d{42, "example"};
   std::cout << d.value() << '\n' << d.msg() << '\n';
}
```

Predict the output from Listing 66-2.

Notice that the visible class is now initialized only once and that the derived class is the one that initializes it. Thus, every class message is "derived". This example is unusual because I want to illustrate how virtual base classes work. Most virtual base classes define only a default constructor. This frees authors of derived classes from concerning themselves with passing arguments to the virtual base class constructor. Instead, every derived class invokes the default constructor; it doesn't matter which class is the most derived.

Figure 66-2 depicts the new class diagram, using virtual inheritance.

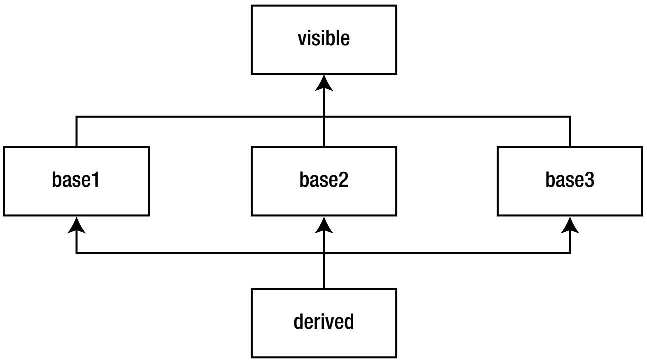


Figure 66-2. *Class diagram with virtual inheritance*

Java-Like Interfaces

Programming with interfaces has some important advantages. Being able to separate interfaces from implementations makes it easy to change implementations without affecting other code. If you have to use interfaces, you can easily do so in C++.

C++ has no formal notion of interfaces, but it supports interface-based programming. The essence of an interface in Java and similar languages is that an interface has no data members, and the member functions have no implementations. Recall from Exploration 38 that such a function is called a *pure virtual function*. Thus, an interface is merely an ordinary class in which you do not define any data members, and you declare all member functions as pure virtual.

For example, Java has the Hashable interface, which defines the hash and equalTo functions. Listing 66-3 shows the equivalent C++ class.

Listing 66-3. The `Hashable` Interface in C++

```
class Hashable
{
public:
   virtual ~Hashable();
   virtual unsigned long hash() const = 0;
   virtual bool equalTo(Hashable const&) const = 0;
};
```

Any class that implements the Hashable interface must override all the member functions. For example, HashableString implements Hashable for a string, as shown in Listing 66-4.

Listing 66-4. The `HashableString` Class

```
class HashableString : public Hashable
{
public:
   HashableString() : string_{} {}
   ~HashableString() override;
   unsigned long hash() const override;
   bool equalTo(Hashable const&) const override;
```

```
    // Implement the entire interface of std::string ...
private:
   std::string string_;
};
```

Note that HashableString does *not* derive from std::string. Instead, it encapsulates a string and delegates all string functions to the string_ object it holds.

The reason you cannot derive from std::string is the same reason Hashable contains a virtual destructor. Recall from Exploration 38 that any class with at least one virtual function should make its destructor virtual. Let me explain the reason.

To understand the problem, think about what would happen if HashableString were to derive from std::string. Suppose that somewhere else in the program is some code that frees strings (maybe a pool of common strings). This code stores strings as std::string pointers. If HashableString derives from std::string, this is fine. But when the pool object frees a string, it calls the std::string destructor. Because this destructor is not virtual, the HashableString destructor never runs, resulting in undefined behavior. Listing 66-5 illustrates this problem.

Listing 66-5. Undefined Behavior Arises from `HashableString` That Derives from `std::string`

```
#include <iostream>
#include <istream>
#include <string>
#include <unordered_set>
#include <utility>

class Hashable
{
public:
   virtual ~Hashable() {}
   virtual unsigned long hash() const = 0;
   virtual bool equalTo(Hashable const&) const = 0;
};

class HashableString : public std::string, public Hashable
{
public:
   HashableString() : std::string{} {}
   HashableString(std::string&& str) : std::string{std::move(str)} {}
   ~HashableString() override {}

   unsigned long hash() const override {
      return std::hash<std::string>()(*this);
   }

   bool equalTo(Hashable const& s) const override {
      return dynamic_cast<HashableString const&>(s) == *this;
   }

};
```

```
class string_pool
{
public:
   string_pool() : pool_{} {}
   ~string_pool() {
      while (not pool_.empty()) {
         std::string* ptr{ *pool_.begin() };
         pool_.erase(pool_.begin());
         delete ptr;
      }
   }
   std::string* add(std::string&& str) {
      std::string* ptr = new std::string{std::move(str)};
      pool_.insert(ptr);
      return ptr;
   }
private:
   std::unordered_set<std::string*> pool_;
};

int main()
{
   string_pool pool{};
   HashableString str{};
   while (std::cin >> str)
   {
      std::cout << "hash of \"" << str << "\" = " << str.hash() << '\n';
      pool.add(std::move(str));
   }
}
```

Similarly, if a pool of Hashable pointers were to delete its contents, the std::string destructor would not run. Again, the behavior is undefined. You can at least expect a memory leak, because the memory allocated for the string contents will never be deleted.

On the other hand, if HashableString does not derive from std::string, how can the string pool manage these hashable strings? The short answer is that it cannot. The long answer is that thinking in terms of Java solutions does not work well in C++, because C++ offers a better solution to this kind of problem: templates.

Interfaces vs. Templates

As you can see, C++ supports Java-style interfaces, but that style of programming can lead to difficulties. There are times when Java-like interfaces are the correct C++ solution. There are other situations, however, when C++ offers superior solutions, such as templates.

Instead of writing a HashableString class, write a hash<> class template and specialize the template for any type that has to be stored in a hash table. The primary template provides the default behavior; specialize hash<> for the std::string type. In this way, the string pool can easily store std::string pointers and destroy the string objects properly, and a hash table can compute hash values for strings (and anything else you have to store in the hash table). Listing 66-6 shows one way to write the hash<> class template and a specialization for std::string.

Listing 66-6. The hash<> Class Template

```
template<class T>
class hash
{
public:
   std::size_t operator()(T const& x) const
   {
     return reinterpret_cast<std::size_t>(&x);
   }
};

template<>
class hash<std::string>
{
public:
   std::size_t operator()(std::string const& str) const
   {
      std::size_t h(0);
      for (auto c : str)
         h = h << 1 | c;
      return h;
   }
};
```

Now try using the hash<> class template to rewrite the string_pool class. Compare your solution with Listing 66-7.

Listing 66-7. Rewriting `string_pool` to Use `hash<>`

```
#include <iostream>
#include <istream>
#include <utility>
#include "hash.hpp"        // Listing 66-6

#include "string_pool.hpp" // Copied from Listing 66-5

int main()
{
   string_pool pool{};
   std::string str{};
   hash<std::string> hasher{};
   while (std::cin >> str)
   {
      std::cout << "hash of \"" << str << "\" = " << hasher(str) << '\n';
      pool.add(std::move(str));
   }
}
```

Use the exact same string_pool class as you did in Listing 66-5. The program that uses the string pool is simple and clear and has the distinct advantage of being well-formed and correct. (By the way, the standard library offers std::hash and specializes it for std::string. Trust your library's implementation to be vastly superior to the toy implementation in this Exploration.)

This approach gives all the functionality of the Hashable interface, but in a manner that allows any type to be hashable without giving up any well-defined behavior. In addition, the hash() function is no longer virtual and can even be an inline function. The speed-up can be considerable if the hash table is accessed in a critical performance path.

Mix-Ins

Another approach to multiple inheritance that you find in languages such as Ruby is the *mix-in*. A mix-in is a class that typically has no data members, although this is not a requirement in C++ (as it is in some languages). Usually, a C++ mix-in is a class template that defines some member functions that call upon the template arguments to provide input values for those functions.

For example, in Exploration 59, you saw a way to implement assignment in terms of a swap member function. This is a useful idiom, so why not capture it in a mix-in class, so you can easily reuse it? The mix-in class is actually a class template that takes a single template argument: the derived class. The mix-in defines the assignment operator, using the swap function that the template argument provides.

Confused yet? You aren't alone. This is a common idiom in C++, but one that takes time before it becomes familiar and natural. Listing 66-8 helps to clarify how this kind of mix-in works.

Listing 66-8. The `assignment_mixin` Class Template

```
template<class T>
class assignment_mixin {
public:
   T& operator=(T rhs)
   {
      rhs.swap(static_cast<T&>(*this));
      return static_cast<T&>(*this);
   }
};
```

The trick is that instead of swapping *this, the mix-in class casts itself to a reference to the template argument, T. In this way, the mix-in never has to know anything about the derived class. The only requirement is that the class, T, must be copyable (so it can be an argument to the assignment function) and have a swap member function.

In order to use the assignment_mixin class, derive your class from the assignment_mixin (as well as any other mix-ins you wish to use), using the derived class name as the template argument. Listing 66-9 shows an example of how a class uses mix-ins.

Listing 66-9. Using `mix-in` Class Template

```
#include <string>
#include <utility>
#include "assignment_mixin.hpp" // Listing 66-8

class thing: public assignment_mixin<thing> {
public:
   thing() : value_{} {}
   thing(std::string&& s) : value_{std::move(s)} {}
   void swap(thing& other) { value_.swap(other.value_); }
private:
   std::string value_;
};
```

```
int main()
{
   thing one{};
   thing two{"two"};
   one = two;
}
```

This C++ idiom is hard to comprehend at first, so let's break it down. First, consider the assignment_mixin class template. Like many other templates, it takes a single template parameter. It defines a single member function, which happens to be an overloaded assignment operator. There's nothing particularly special about assignment_mixin.

But assignment_mixin has one important property: the compiler can compile the template even if the template argument is an incomplete class. The compiler doesn't have to expand the assignment operator until it is used, and at that point, T must be complete. But for the class itself, T can be incomplete. If the mix-in class were to declare a data member of type T, then the compiler would require that T be a complete type when the mix-in is instantiated, because it would have to know the size of the mix-in.

In other words, you can use assignment_mixin as a base class, even if the template argument is an incomplete class.

When the compiler processes a class definition, immediately upon seeing the class name, it records that name in the current scope as an incomplete type. Thus, when assignment_mixin<thing> appears in the base class list, the compiler is able to instantiate the base class template using the incomplete type, thing, as the template argument.

By the time the compiler gets to the end of the class definition, thing becomes a complete type. After that, you will be able to use the assignment operator, because when the compiler instantiates that template, it needs a complete type, and it has one.

Protected Access Level

In addition to the private and public access levels, C++ offers the protected access level. A protected member is accessible only to the class itself and to derived classes. To all other would-be users, a protected member is off-limits, just like private members.

Most members are private or public. Use protected members only when you are designing a class hierarchy and you deliberately want derived classes to call a certain member function but don't want anyone else to call it.

Mix-in classes sometimes have a protected constructor. This ensures that no one tries to construct a stand-alone instance of the class. Listing 66-10 shows assignment_mixin with a protected constructor.

Listing 66-10. Adding a Protected Constructor to the `assignment_mixin` Class Template

```
template<class T>
class assignment_mixin {
public:
   T& operator=(T rhs)
   {
      rhs.swap(static_cast<T&>(*this));
      return static_cast<T&>(*this);
   }
protected:
  assignment_mixin() {}
};
```

Multiple inheritance also appears in the C++ standard library. You know about istream for input and ostream for output. The library also has iostream, so a single stream can perform input and output. As you might expect, iostream derives from istream and ostream. The only quirk has nothing to do with multiple inheritance: iostream is defined in the <istream> header. The <iostream> header defines the names std::cin, std::cout, etc. The header name is an accident of history.

The next Exploration continues your advanced study of types, by looking at policies and traits.

EXPLORATION 67

Traits and Policies

Although you may still be growing accustomed to templates, it's time to explore two common, related use patterns: traits and policies. Programming with traits and policies is probably a new style for you, but it is common in C++. As you will discover in this Exploration, this technique is extremely flexible and powerful. Traits and policies underlie much of the C++ standard library. This Exploration looks at some of the traits and policies in the standard library, so that you can learn how to take advantage of them. It then helps you take the first steps toward writing your own.

Case Study: Iterators

Consider the humble iterator. Consider the `std::advance` function (Exploration 44). The `advance` function changes the position to which an iterator points. The `advance` function knows nothing about container types; it knows only about iterators. Yet somehow, it knows that if you try to advance a `vector`'s iterator, it can do so simply by adding an integer to the iterator. But if you advance a `list`'s iterator, the `advance` function must step the iterator one position at a time until it arrives at the desired destination. In other words, the `advance` function implements the optimal algorithm for changing the iterator's position. The only information available to the `advance` function must come from the iterators themselves, and the key piece of information is the iterator kind. In particular, only random access iterators permit rapid advancement via addition. All other iterators must follow the step-by-step approach. So how does `advance` know what kind of iterator it has?

In most OOP languages, an iterator would derive from a common base class, which would implement a virtual `advance` function. The `advance` algorithm would call that virtual function and let normal object-oriented dispatching take care of the details. C++ certainly could take that approach, but it doesn't.

Instead, C++ uses a technique that does not require looking up a virtual function and making an extra function call. Rather, C++ uses a technique that does not force you to derive all iterators from a single base class. If you implement a new container, you get to pick the class hierarchy. C++ provides the `std::iterator` base class template, which you can use if you want, but you don't have to use it. Instead, the `advance` algorithm (and all other code that uses iterators) relies on a traits template.

Traits are attributes or properties of a type. In this case, an iterator's traits describe the iterator kind (random access, bidirectional, forward, input, or output), the type that the iterator points to, and so on. The author of an iterator class specializes the `std::iterator_traits` class template to define the traits of the new iterator class. Iterator traits make more sense with an example, so let's take a look at Listing 67-1, which shows one possible implementation of `std::advance`.

Listing 67-1. One Possible Implementation of `std::advance`

```
#include <iostream>
#include <iterator>
#include <ostream>
#include <string>
```

```
void trace(std::string const& msg)
{
   std::cout << msg << '\n';
}

// Default implementation: advance the slow way
template<class Kind>
class iterator_advancer
{
public:
   template<class InIter, class Distance>
   void operator()(InIter& iter, Distance distance)
   {
      trace("iterator_advancer<>");
      for ( ; distance != 0; --distance)
         ++iter;
   }
};

// Partial specialization for bi-directional iterators
template<>
class iterator_advancer<std::bidirectional_iterator_tag>
{
public:
   template<class BiDiIter, class Distance>
   void operator()(BiDiIter& iter, Distance distance)
   {
      trace("iterator_advancer<bidirectional_iterator_tag>");
      if (distance < 0)
         for ( ; distance != 0; ++distance)
            --iter;
      else
         for ( ; distance != 0; --distance)
            ++iter;
   }
};

template<class InIter, class Distance>
void my_advance(InIter& iter, Distance distance)
{
    iterator_advancer<typename std::iterator_traits<InIter>::iterator_category>{}

        (iter, distance);
}
```

This code is not as difficult to understand as it appears. The `iterator_advancer` class template provides one function, the function call operator. The implementation advances an input iterator `distance` times. This implementation works for a non-negative `distance` with any kind of iterator.

Bidirectional iterators permit a negative value for `distance`. Partial template specialization permits a separate implementation of `iterator_advancer` just for bidirectional iterators. The specialization checks whether `distance` is negative; negative and non-negative values are handled differently.

■ **Note** Remember from Exploration 51 that only classes can use partial specialization. That's why `iterator_advancer` is a class template with a function call operator instead of a function template. This idiom is common in C++. Another approach is to use function overloading, passing the iterator kind as an additional argument. The value is unimportant, so just pass a default-constructed object. Advanced overloading will be covered in Exploration 69.

Let's assemble the pieces. The `my_advance` function creates an instance of `iterator_advancer` and calls its function call operator, passing the `iter` and `distance` arguments. The magic is the `std::iterator_traits` class template. This class template has a few member typedefs, including `iterator_category`.

All bidirectional iterators must define the member typedef, `iterator_category`, as `std::bidirectional_iterator_tag`. Thus, when your program calls `my_advance`, and passes a bidirectional iterator (such as a `std::list` iterator) as the first argument, the `my_advance` function queries `iterator_traits` to discover the `iterator_category`. The compiler uses template specialization to decide which implementation of `iterator_advancer` to choose. The compiler then generates the code to call the correct function. The compiler takes care of all this magic—your program pays no runtime penalty.

Now try running the program in Listing 67-2 to see which `iterator_advancer` specialization is called in each situation.

Listing 67-2. Example Program to Use the `my_advance` Function

```
#include <fstream>
#include <iostream>
#include <istream>
#include <iterator>
#include <list>
#include <ostream>
#include <string>
#include <vector>

#include "advance.hpp" // Listing 67-1

int main()
{
   std::vector<int> vector{ 10, 20, 30, 40 };
   std::list<int> list(vector.begin(), vector.end());
   std::vector<int>::iterator vector_iterator{vector.begin()};
   std::list<int>::iterator list_iterator{list.begin()};
   std::ifstream file{"advance.hpp"};
   std::istream_iterator<std::string> input_iterator{file};

   my_advance(input_iterator, 2);
   my_advance(list_iterator, 2);
   my_advance(vector_iterator, 2);
}
```

Notice any problems? **What kind of iterator does a vector use?** ________________ **Which specialization does the compiler pick?** ________________ The compiler does not follow class hierarchies when picking template specializations, so the fact that `random_access_iterator_tag` derives from `bidirectional_iterator_tag` is irrelevant in this case. If you want to specialize `iterator_advancer` for random access iterators, you must provide another specialization, this time for `random_access_iterator_tag`. Remember from Exploration 44 that random access iterators permit arithmetic on iterators. Thus, you can advance an iterator rapidly by adding the distance. **Implement a partial specialization of `iterator_advancer` for random access iterators**.

Compare your solution with the snippet in Listing 67-3.

Listing 67-3. Specializing `iterator_advancer` for Random Access Iterators

```
// Partial specialization for random access iterators
template<>
class iterator_advancer<std::random_access_iterator_tag>
{
public:
   template<class RandomIter, class Distance>
   void operator()(RandomIter& iter, Distance distance)
   {
      trace("iterator_advancer<random_access_iterator_tag>");
      iter += distance;
   }
};
```

Now rerun the sample program to see that the compiler picks the correct specialization.

A good optimizing compiler can take the `my_advance` function with the random-access specialization of `iterator_advancer` and easily compile optimal code, turning a call to the `my_advance` function into a single addition, with no function-call overhead. In other words, the layers of complexity that traits and policies introduce do not necessarily equate to bloated code and poor runtime performance. The complexity is conceptual, and once you understand what the traits do and how they work, you can let them abstract away the underlying complexity. They will make your job easier.

Iterator Traits

The class template, `iterator_traits` (defined in `<iterator>`), is an example of a *traits* type. A traits type provides traits, or characteristics, of another type. In this case, `iterator_traits` informs you about several traits of an iterator exposed via typedefs.

- **difference_type:** A signed integer type that represents the difference between two iterators. If you have two iterators that point into the same container, the `distance` function returns the distance between them—that is, the number of positions that separate the iterators. If the iterators are bidirectional or random access, the distance can be negative.
- **iterator_category:** The iterator kind, which must be one of the following types (also defined in `<iterator>`):
    - `bidirectional_iterator_tag`
    - `forward_iterator_tag`
    - `input_iterator_tag`
    - `output_iterator_tag`
    - `random_access_iterator_tag`

Some of the standard algorithms use template specialization and the `iterator_category` to provide optimal implementations for different kinds of iterators.

- **pointer:** A typedef that represents a pointer to a value.
- **reference:** A typedef that represents a reference to a value.
- **value_type:** The type of values to which the iterator refers.

Can you think of another traits type in the standard library? ________________ The first one that I introduced in Exploration 2 is `std::numeric_limits` (Exploration 25). Another traits class that I've mentioned without explanation is `std::char_traits` (defined in `<string>`). But most of the interesting traits are the type traits.

Type Traits

The `<type_traits>` header defines a suite of traits templates that describe the characteristics of a type. They range from simple queries, such as `std::is_integral<>`, which tells you whether a type is one of the built-in integral types, to more sophisticated queries, such as `std::is_nothrow_move_constructible<>`, which tells you whether a class has a `noexcept` move constructor. Some traits modify types, such as `std::remove_reference<>`, which transforms `int&` to `int`, for example.

The `std::move()` function uses type traits, just to name one use of type traits in the standard library. Remember that all it does is change an lvalue to an rvalue. It uses `remove_reference` to strip the reference from its argument and then adds `&&` to turn the result into an rvalue reference, as follows:

```
template<class T>
typename std::remove_reference<T>::type&& move(T&& t) noexcept
{
   return static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

Notice the use of the `type` member typedef. That is how the type traits expose the result of their transformation. The query traits declare `type` to be a typedef for `std::true_type` or `std::false_type`; these classes declare a `value` member to be `true` or `false` at compile time. Although you can create an instance of `true_type` or `false_type` and evaluate them at runtime, the typical use is to use them to specialize a template.

The `<type_traits>` header has much more to offer, and Exploration 70 will delve deeper into its mysteries. For now, let's tackle a more mundane traits class, `std::char_traits`.

Case Study: char_traits

Among the difficulties in working with characters in C++ is that the `char` type may be signed or unsigned. The size of a `char` relative to the size of an `int` varies from compiler to compiler. The range of valid character values also varies from one implementation to another and can even change while a program is running. A time-honored convention is to use `int` to store a value that may be a `char` or a special value that marks end-of-file, but nothing in the standard supports this convention. You may need to use `unsigned int` or `long`.

In order to write portable code, you need a traits class to provide a typedef for the integer type to use, the value of the end-of-file marker, and so on. That's exactly what `char_traits` is for. When you use `std::char_traits<char>::int_type`, you know you can safely store any `char` value or the end-of-file marker (which is `std::char_traits<char>::eof()`).

The standard `istream` class has a `get()` function that returns an input character or the special end-of-file marker when there is no more input. The standard `ostream` class offers `put(c)` to write a character. **Use these functions with `char_traits` to write a function that copies its standard input to its standard output, one character at a time.** Call `eof()` to obtain the special end-of-file value and `eq_int_type(a,b)` to compare two integer representations of characters for equality. Both functions are static member functions of the `char_traits` template, which you must instantiate with the desired character type. Call `to_char_type` to convert the integer representation back to a `char`. Compare your solution with Listing 67-4.

Listing 67-4. Using Character Traits When Copying Input to Output

```
#include <iostream>
#include <istream>
#include <ostream>
#include <string>        // for char_traits

int main()
{
   typedef std::char_traits<char> char_traits; // for brevity and clarity
   char_traits::int_type c{};
   while (c = std::cin.get(), not char_traits::eq_int_type(c, char_traits::eof()))
      std::cout.put(char_traits::to_char_type(c));
}
```

First, notice the loop condition. Recall from Exploration 46 that the comma can separate two expressions; the first sub-expression is evaluated, then the second. The result of the entire expression is the result of the second sub-expression. In this case, the first sub-expression assigns `get()` to `c`, and the second sub-expression calls `eq_int_type`, so the result of the loop condition is the return value from `eq_int_type`, testing whether the result of `get`, as stored in `c`, is equal to the end-of-file marker. Another way to write the loop condition is as follows:

```
not char_traits::eq_int_type(c = std::cin.get(), char_traits::eof())
```

I don't like to bury assignments in the middle of an expression, so I prefer to use the comma operator in this case. Other developers have a strong aversion to the comma operator. They prefer the embedded assignment style. Another solution is to use a `for` loop instead of a `while` loop, as follows:

```
for (char_traits::int_type c = std::cin.get();
      not char_traits::eq_int_type(c, char_traits::eof());
      c = std::cin.get())
```

The `for`-loop solution has the advantage of limiting the scope of the variable, `c`. But it has the disadvantage of repeating the call to `std::cin.get()`. Any of these solutions is acceptable; pick a style and stick with it.

In this case, `char_traits` seems to make everything more complicated. After all, comparing two integers for equality is easier and clearer when using the == operator. On the other hand, using a member function gives the library-writer the opportunity for added logic, such as checking for invalid character values.

In theory, you could write a `char_traits` specialization that, for instance, implements case-insensitive comparison. In that case, the `eq()` (which compares two characters for equality) and `eq_int_type()` functions would certainly require extra logic. On the other hand, you learned in Exploration 19 that such a traits class cannot be written for many international character sets, at least not without knowing the locale.

In the real world, specializations of `char_traits` are rare.

The `char_traits` class template is interesting nonetheless. A pure traits class template would implement only typedef members, static data members, and sometimes a member function that returns a constant, such as `char_traits::eof()`. Functions such as `eq_int_type()` are not traits, which describe a type. Instead, they are policy functions. A policy class template contains member functions that specify behavior, or policies. The next section looks at policies.

Policy-Based Programming

A *policy* is a class or class template that another class template can use to customize its behavior. The line between traits and policy is fuzzy, but to me, traits are static characteristics and policies are dynamic behavior. In the standard library, the string and stream classes use the char_traits policy class template to obtain type-specific behavior for comparing characters, copying character arrays, and more. The standard library provides policy implementations for the char and wchar_t types.

Suppose you are trying to write a high-performance server. After careful design, implementation, and testing, you discover that the performance of std::string introduces significant overhead. In your particular application, memory is abundant, but processor time is at a premium. Wouldn't it be nice to be able to flip a switch and change your std::string implementation from one that is optimized for space into one that is optimized for speed? Instead, you must write your own string replacement that meets your needs. In writing your own class, you end up rewriting the many member functions, such as find_first_of, that have nothing to do with your particular implementation but are essentially the same for most string implementations. What a waste of time.

Imagine how simple your job would be if you had a string class template that took an extra template argument with which you could select a storage mechanism for the string, substituting memory-optimized or processor-optimized implementations according to your needs. That, in a nutshell, is what policy-based programming is all about.

A possible implementation of std::string is to keep a small character array in the string object for small strings and use dynamic memory allocation for larger strings. In order to conform to the C++ standard, these implementations cannot offer up a menu of policy template arguments that would let you pick the size of the character array. So let us free ourselves from this limitation and write a class that implements all the members of the std::string class but breaks the standard interface by adding a policy template argument. For the sake of simplicity, this book implements only a few functions. Completing the interface of std::string is left as an exercise for the reader. Listing 67-5 shows the new string class template and a few of its member functions. Take a look, and you can see how it takes advantage of the Storage policy.

Listing 67-5. The newstring Class Template

```
#include <algorithm>
#include <cstddef>

template<class Char, class Storage, class Traits = std::char_traits<Char>>
class newstring {
public:
   typedef Char value_type;
   typedef std::size_t size_type;
   typedef typename Storage::iterator iterator;
   typedef typename Storage::const_iterator const_iterator;

   newstring() : storage_() {}
   newstring(newstring&&) = default;
   newstring(newstring const&) = default;
   newstring(Storage const& storage) : storage_(storage) {}
   newstring(Char const* ptr, size_type size) : storage_() {
      resize(size);
      std::copy(ptr, ptr + size, begin());
   }

   static const size_type npos = static_cast<size_type>(-1);

   newstring& operator=(newstring const&) = default;
   newstring& operator=(newstring&&) = default;
   void swap(newstring& str) { storage_.swap(str.storage_); }

   Char operator[](size_type i) const { return *(storage_.begin() + i); }
   Char& operator[](size_type i)      { return *(storage_.begin() + i); }

   void resize(size_type size, Char value = Char()) {
     storage_.resize(size, value);
   }
   void reserve(size_type size)    { storage_.reserve(size); }
   size_type size() const noexcept { return storage_.end() - storage_.begin(); }
   size_type max_size() const noexcept { return storage_.max_size(); }
   bool empty() const noexcept     { return size() == 0; }
   void clear()                    { resize(0); }
   void push_back(Char c)          { resize(size() + 1, c); }

   Char const* c_str() const { return storage_.c_str(); }
   Char const* data() const  { return storage_.c_str(); }

   iterator begin()              { return storage_.begin(); }
   const_iterator begin() const  { return storage_.begin(); }
   const_iterator cbegin() const { return storage_.begin(); }
   iterator end()                { return storage_.end(); }
   const_iterator end() const    { return storage_.end(); }
   const_iterator cend() const   { return storage_.end(); }

   size_type find(newstring const& s, size_type pos = 0) const {
      pos = std::min(pos, size());
      const_iterator result( std::search(begin() + pos, end(),
                             s.begin(), s.end(), Traits::eq) );
      if (result == end())
         return npos;
      else
         return result - begin();
   }

private:
   Storage storage_;
};

template<class Traits>
class newstringcmp
{
public:
   bool operator()(typename Traits::value_type a, typename Traits::value_type b)
   const
   {
      return Traits::cmp(a, b) < 0;
   }
};
```

```
template<class Char, class Storage1, class Storage2, class Traits>
bool operator <(newstring<Char, Storage1, Traits> const& a,
                newstring<Char, Storage2, Traits> const& b)
{
   return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end(),
                                       newstringcmp<Traits>());
}

template<class Char, class Storage1, class Storage2, class Traits>
bool operator ==(newstring<Char, Storage1, Traits> const& a,
                 newstring<Char, Storage2, Traits> const& b)
{
   return std::equal(a.begin(), a.end(), b.begin(), b.end(), Traits::eq);
}
```

The newstring class relies on Traits for comparing characters and Storage for storing them. The Storage policy must provide iterators for accessing the characters themselves and a few basic member functions (data, max_size, reserve, resize, swap), and the newstring class provides the public interface, such as the assignment operator and search member functions.

Public comparison functions use standard algorithms and Traits for comparisons. Notice how the comparison functions require their two operands to have the same Traits (otherwise, how could the strings be compared in a meaningful way?) but allow different Storage. It doesn't matter how the strings store their contents if you want to know only whether two strings contain the same characters.

The next step is to write some storage policy templates. The storage policy is parameterized on the character type. The simplest Storage is vector_storage, which stores the string contents in a vector. Recall from Exploration 62, that a C character string ends with a null character. The c_str() member function returns a pointer to a C-style character array. In order to simplify the implementation of c_str, the vector stores a trailing null character after the string contents. Listing 67-6 shows part of an implementation of vector_storage. You can complete the implementation on your own.

Listing 67-6. The vector_storage Class Template

```
#include <vector>

template<class Char>
class vector_storage {
public:
   typedef std::size_t size_type;
   typedef Char value_type;
   typedef typename std::vector<Char>::iterator iterator;
   typedef typename std::vector<Char>::const_iterator const_iterator;

   vector_storage() : string_(1, Char()) {}

   void swap(vector_storage& storage) { string_.swap(storage.string_); }
   size_type max_size() const { return string_.max_size() - 1; }
   void reserve(size_type size) { string_.reserve(size + 1); }
   void resize(size_type newsize, value_type value) {
      // if the string grows, overwrite the null character, then resize
      if (newsize >= string_.size()) {
         string_[string_.size() - 1] = value;
         string_.resize(newsize + 1, value);
      }
      else
         string_.resize(newsize + 1);
      string_[string_.size() - 1] = Char();
   }
   Char const* c_str() const { return &string_[0]; }

   iterator begin()             { return string_.begin(); }
   const_iterator begin() const { return string_.begin(); }
   // Skip over the trailing null character at the end of the vector
   iterator end()               { return string_.end() - 1; }
   const_iterator end() const   { return string_.end() - 1; }

private:
   std::vector<Char> string_;
};
```

The only difficulty in writing `vector_storage` is that the vector stores a trailing null character, so the `c_str` function can return a valid C-style character array. Therefore, the end function has to adjust the iterator that it returns.

Another possibility for a storage policy is `array_storage`, which is just like `vector_storage`, except it uses an `array`. By using an array, all storage is local. The array size is the maximum capacity of the string, but the string size can vary up to that maximum. **Write `array_storage`.** Compare your result with mine in Listing 67-7.

Listing 67-7. The `array_storage` Class Template

```
#include <algorithm>
#include <cstdlib>
#include <stdexcept>
#include <array>

template<class Char, std::size_t MaxSize>
class array_storage {
public:
   typedef std::array<Char, MaxSize> array_type;
   typedef std::size_t size_type;
   typedef Char value_type;
   typedef typename array_type::iterator iterator;
   typedef typename array_type::const_iterator const_iterator;

   array_storage() : size_(0), string_() { string_[0] = Char(); }

   void swap(array_storage& storage) {
      string_.swap(storage.string_);
      std::swap(size_, storage.size_);
   }
   size_type max_size() const { return string_.max_size() - 1; }
   void reserve(size_type size) {
     if (size > max_size()) throw std::length_error("reserve");
   }
   void resize(size_type newsize, value_type value) {
      if (newsize > max_size())
         throw std::length_error("resize");
      if (newsize > size_)
         std::fill(begin() + size_, begin() + newsize, value);
      size_ = newsize;
      string_[size_] = Char();
   }
   Char const* c_str() const { return &string_[0]; }

   iterator begin()             { return string_.begin(); }
   const_iterator begin() const { return string_.begin(); }
   iterator end()               { return begin() + size_; }
   const_iterator end() const   { return begin() + size_; }

private:
   size_type size_;
   array_type string_;
};
```

One difficulty when writing new string classes is that you must write new I/O functions too. Unfortunately, this takes a fair bit of work and a solid understanding of the stream class templates and stream buffers. Handling padding and field adjustment is easy, but there are subtleties to the I/O streams that I have not covered, such as integration with C stdio, tying input and output streams so that prompts appear before the user is asked for input, etc. So just copy my solution in Listing 67-8 into `newstring.hpp`.

Listing 67-8. Output Function for `newstring`

```
template<class Char, class Storage, class Traits>
std::basic_ostream<Char, Traits>&
  operator<<(std::basic_ostream<Char, Traits>& stream,
             newstring<Char, Storage, Traits> const& string)
{
   typename std::basic_ostream<Char, Traits>::sentry sentry{stream};
   if (sentry)
   {
      bool needs_fill{stream.width() != 0 and string.size() > stream.width()};
      bool is_left_adjust{
         (stream.flags() & std::ios_base::adjustfield) == std::ios_base::left };
      if (needs_fill and not is_left_adjust)
      {
         for (std::size_t i{stream.width() - string.size()}; i != 0; --i)
            stream.rdbuf()->sputc(stream.fill());
      }
      stream.rdbuf()->sputn(string.data(), string.size());
      if (needs_fill and is_left_adjust)
      {
         for (std::size_t i{stream.width() - string.size()}; i != 0; --i)
            stream.rdbuf()->sputc(stream.fill());
      }
   }
   stream.width(0);
   return stream;
}
```

The sentry class manages some bookkeeping on behalf of the stream. The output function handles padding and adjustment. If you are curious about the details, consult a good reference.

The input function also has a sentry class, which skips leading white space on your behalf. The input function has to read characters until it gets to another whitespace character or the string fills or the width limit is reached. See Listing 67-9 for my version.

Listing 67-9. Input Function for newstring

```
template<class Char, class Storage, class Traits>
std::basic_istream<Char, Traits>&
  operator>>(std::basic_istream<Char, Traits>& stream,
             newstring<Char, Storage, Traits>& string)
{
   typename std::basic_istream<Char, Traits>::sentry sentry{stream};
   if (sentry)
   {
      std::ctype<Char> const& ctype(
         std::use_facet<std::ctype<Char>>(stream.getloc()) );
      std::ios_base::iostate state{ std::ios_base::goodbit };
      std::size_t max_chars{ string.max_size() };
      if (stream.width() != 0 and stream.width() < max_chars)
         max_chars = stream.width();
      string.clear();
      while (max_chars-- != 0) {
         typename Traits::int_type c{ stream.rdbuf()->sgetc() };
         if (Traits::eq_int_type(Traits::eof(), c)) {
            state |= std::ios_base::eofbit;
            break; // is_eof
         }
         else if (ctype.is(ctype.space, Traits::to_char_type(c)))
            break;
         else {
            string.push_back(Traits::to_char_type(c));
            stream.rdbuf()->sbumpc();
         }
      }
      if (string.empty())
         state |= std::ios_base::failbit;
      stream.setstate(state);
      stream.width(0);
   }
   return stream;
}
```

The break statement exits a loop immediately. You are probably familiar with this statement or something similar. Experienced programmers may be surprised that no example has required this statement until now. One reason is that I gloss over error-handling, which would otherwise be a common reason to break out of a loop. In this case, when the input reaches end of file or white space, it is time to exit the loop. The partner to break is continue, which immediately reiterates the loop. In a for loop, continue evaluates the iterate part of the loop header and then the condition. I have rarely needed to use continue in real life and could not think of any reasonable example that uses continue, but I mention it only for the sake of completeness.

As you know, the compiler finds your I/O operators by matching the type of the right-hand operand, `newstring`, with the type of the function parameter. In this simple case, you can easily see how the compiler performs the matching and finds the right function. Throw some namespaces into the mix, and add some type conversions, and everything gets a little bit more muddled. The next Exploration delves more closely into namespaces and the rules that the C++ compiler applies in order to find your overloaded function names (or not find them, and, therefore, how to fix that problem).

EXPLORATION 68

Names, Namespaces, and Templates

The basics of using namespaces and templates are straightforward and easy to learn. Taking advantage of argument-dependent lookup (ADL) is also simple: declare free functions and operators in the same namespace as your classes. But sometimes life isn't so simple. Especially when using templates, you can get stuck in strange corners, and the compiler issues bizarre and useless messages, and you realize you should have spent more time studying names, namespaces, and templates beyond the basics.

The detailed rules can be excruciatingly complicated, because they must cover all the pathological cases, for example, to explain why the following is legal (albeit resulting in some compiler warnings) and what it means:

```
enum X { X };
void XX(enum X X=::X) { if (enum X X=X) X; }
```

and why the following is illegal:

```
enum X { X } X;
void XX(X X=X) { if (X X=X) X; }
```

But rational programmers don't write code in that manner, and some commonsense guidelines go a long way toward simplifying the complicated rules. Thus, this Exploration provides more details than earlier in the book but omits many picky details that matter only to entrants in obfuscated C++ contests.

Common Rules

Certain rules apply to all types of name lookup. (Subsequent sections will examine the rules particular to certain contexts). The basic rule is that the compiler must know what a name means when it sees the name in source code.

Most names must be declared earlier in the file (or in an included header) than where the name is used. The only exception is in the definition of a member function: a name can be another member of the same class, even if the member is declared later in the class definition than the use of that name. Names must be unique within a scope, except for overloaded functions and operators. The compiler issues an error if you attempt to declare two names that would conflict, such as two variables with the same name in the same scope, or a data member and member function with the same name in a single class.

Functions can have multiple declarations with the same name, following the rules for overloading, that is, argument number or types must be different, or `const` qualification must be different for member functions.

Access rules (public, private, or protected) have no effect on name lookup rules for members of a class (nested types and typedefs, data members, and member functions). The usual name lookup rules identify the proper declaration, and only then does the compiler check whether access to the name is permitted.

Whether a name is that of a virtual function has no effect on name lookup. The name is looked up normally, and once the name is found, then the compiler checks whether the name is that of a virtual function. If so, and if the object is accessed via a reference or pointer, the compiler generates the necessary code to perform a runtime lookup of the actual function.

Name Lookup in Templates

Templates complicate name lookup. In particular, a name can depend on a template parameter. Such a dependent name has different lookup rules than nondependent names. Dependent names can change meaning according to the template arguments used in a template specialization. One specialization may declare a name as a type, and another may declare it as a function. (Of course, such a programming style is highly discouraged, but the rules of C++ must allow for such possibilities.)

Lookup of nondependent names follows the usual name lookup rules where the template is defined. Lookup of dependent names may include lookup in namespaces associated with the template instantiation, in addition to the normal rules that apply where the template is defined. Subsequent sections provide additional details, according to the kind of name lookup being performed.

Three Kinds of Name Lookup

C++ defines three kinds of name lookup: member access operators; names prefaced by class, enumeration, or namespace names; and bare names.

- A class member access operator is `.` or `->`. The left-hand side is an expression that yields an object of class type, reference to an object, or pointer to an object. The dot (.) requires an object or reference, and `->` requires a pointer. The right-hand side is a member name (data member or member function). For example, in the expression `cout.width(3)`, `cout` is the object, and `width` is the member name.
- A class, enumeration, or namespace name may be followed by the scope operator (`::`) and a name, such as `std::string`. The name is said to be *qualified* by the class, enumeration, or namespace name. No other kind of name may appear to the left of the scope operator. The compiler looks up the name in the scope of the class, enumeration, or namespace. The name itself may be another class, enumerator, or namespace name, or it may be a function, variable, or typedef name. For example, in `std::chrono::system_clock::duration`, `std` and `chrono` are namespace names; `sytem_clock` is a class name; and `duration` is a member typedef.
- A plain identifier or operator is called an *unqualified* name. The name can be a namespace name, type name, function name, or object name, depending on context. Different contexts have slightly different lookup rules.

The next three sections describe each style of name lookup in more detail.

Member Access Operators

The simplest rules are for member access operators. The left-hand side of the member access operator (. or ->) determines the context for the lookup. The object must have class type (or pointer or reference to a class type), and the name on the right-hand side must be a data member or member function of that class or of an ancestor class. The search begins with the declared type of the object and continues with its base class (or classes, searching multiple classes from left to right, in order of declaration), and their base classes, and so on, stopping at the first class with a matching name.

If the name is a function, the compiler collects all declarations of the same name in the same class and chooses one function according to the rules of function and operator overloading. Note that the compiler does not consider any functions in ancestor classes. The name lookup stops as soon as the name is found. If you want a base class's names to participate in operator overloading, use a *using* declaration in the derived class to bring the base class names into the derived class context.

In the body of a member function, the left-hand object can be the `this` keyword, which is a pointer to the object on the left-hand side of the member access operator. If the member function is declared with the `const` qualifier, `this` is a pointer to `const`. If a base class is a template parameter or depends on a template parameter, the compiler does not know which members may be inherited from the base class until the template is instantiated. You should use `this->` to access an inherited member, to tell the compiler that the name is a member name, and the compiler will look up the name when it instantiates the template.

Listing 68-1 demonstrates several uses for member access operators.

Listing 68-1. Member Access Operators

```
#include <cmath>
#include <iostream>

template<class T>
class point2d {
public:
   point2d(T x, T y) : x_(x), y_(y) {}
   virtual ~point2d() {}
   T x() const { return x_; }
   T y() const { return y_; }
   T abs() const { return std::sqrt(x() * x() + y() * y()); }
   virtual void print(std::ostream& stream) const {
      stream << '(' << x() << ", " << y() << ')';
   }
private:
   T x_, y_;
};

template<class T>
class point3d : public point2d<T> {
public:
   point3d(T x, T y, T z) : point2d<T>(x, y), z_(z) {}
   T z() const { return z_; }
   T abs() const {
      return static_cast<T>(std::sqrt(this->x() * this->x() +
                 this->y() * this->y() +
                 this->z() * this->z()));
   }
   virtual void print(std::ostream& stream) const {
      stream << '(' << this->x() << ", " << this->y() << ", " << z() << ')';
   }
private:
   T z_;
};
```

```
template<class T>
std::ostream& operator<<(std::ostream& stream, point2d<T> const& pt)
{
   pt.print(stream);
   return stream;
}

int main()
{
   point3d<int> origin{0, 0, 0};
   std::cout << "abs(origin) = " << origin.abs() << '\n';

   point3d<int> unit{1, 1, 1};
   point2d<int>* ptr{ &unit };
   std::cout << "abs(unit) = " << ptr->abs() << '\n';
   std::cout << "*ptr = " << *ptr << '\n';
}
```

The `main()` function uses member name lookup the way you are used to seeing it. This usage is simple to understand, and you have been using it throughout this book. The use of member name lookup in `point3d::abs()`, however, is more interesting. The use of `this->` is required, because the base class, `point2d<T>` depends on the template parameter, `T`.

The `operator<<` function takes a reference to a `point2d` instance and calls its `print` function. The virtual function is dispatched to the real function, which in this case is `point3d::print`. You know how this works, so this is just a reminder of how the compiler looks up the name `print` in the `point2d` class template, because that is the type of the `pt` function parameter.

Qualified Name Lookup

A *qualified* name uses the scope (`::`) operator. You have been using qualified names from the very first program. The name `std::string` is qualified, which means the name `string` is looked up in a context specified by the `std::` qualifier. In this simple class, `std` names a namespace, so `string` is looked up in that namespace.

The qualifier can also be a class name or the name of a scoped enumeration. Class names can nest, so the left- and right-hand side of the scope operator may be a class name. If the left-hand name is that of an enumerated type, the right-hand name must be an enumerator in that type.

The compiler starts its search with the left-most name. If the left-most name starts with a scope operator (e.g., `::std::string`), the compiler looks up that name in the global scope. Otherwise, it uses the usual name lookup rules for unqualified names (as described in the next section) to determine the scope that it will use for the right-hand side of the scope operator. If the right-hand name is followed by another scope operator, the identified name must be that of a namespace, class, or scoped enumeration, and the compiler looks up the right-hand name in that scope. The process repeats until the compiler has looked up the right-most name.

Within a namespace, a *using* directive tells the compiler to search in the target namespace as well as the namespace that contains the *using* directive. In the following example, the qualified name `ns2::Integer` tells the compiler to search in namespace `ns2` for name `Integer`. Because `ns2` contains a *using* directive, the compiler also searches in namespace `ns1`, and so finds the `Integer` typedef.

```
namespace ns1 { typedef int Integer; }
namespace ns2 { using namespace ns1; }
namespace ns3 { ns2::Integer x; }
```

A *using* declaration is slightly different. A *using* directive affects which namespaces the compiler searches to find a name. A *using* declaration doesn't change the set of namespaces to search but merely adds one name to the containing namespace. In the following example, the *using* declaration brings the name `Integer` into namespace `ns2`, just as though the typedef were written in `ns2`.

```
namespace ns1 { typedef int Integer; }
namespace ns2 { using ns1::Integer; }
namespace ns3 { ns2::Integer x; }
```

When a name depends on a template parameter, the compiler must know whether the name is that of a type or something else (function or object), because it affects how the compiler parses the template body. Because the name is dependent, it could be a type in one specialization and a function in another. So you have to tell the compiler what to expect. If the name should be a type, preface the qualified name with the keyword `typename`. Without the `typename` keyword, the compiler assumes the name is that of a function or object. You need the `typename` keyword with a dependent type, but it doesn't hurt if you provide it before a nondependent type.

Listing 68-2 shows several examples of qualified names.

Listing 68-2. Qualified Name Lookup

```
#include <chrono>
#include <iostream>

namespace outer {
   namespace inner {
      class base {
      public:
         int value() const { return 1; }
         static int value(long x) { return static_cast<int>(x); }
      };
   }

   template<class T>
   class derived : public inner::base {
   public:
      typedef T value_type;
      using inner::base::value;
      static value_type example;
      value_type value(value_type i) const { return i * example; }
   };

   template<class T>
   typename derived<T>::value_type derived<T>::example = 2;
}

template<class T>
class more_derived : public outer::derived<T>{
public:
   typedef outer::derived<T> base_type;
   typedef typename base_type::value_type value_type;
   more_derived(value_type v) : value_{this->value(v)} {}
   value_type get_value() const { return value_; }
private:
   value_type value_;
};
```

```
int main()
{
   std::chrono::system_clock::time_point now{std::chrono::system_clock::now()};
   std::cout << now.time_since_epoch().count() << '\n';

   outer::derived<int> d;
   std::cout << d.value() << '\n';
   std::cout << d.value(42L) << '\n';
   std::cout << outer::inner::base::value(2) << '\n';

   more_derived<int> md(2);
   std::cout << md.get_value() << '\n';
}
```

The standard chrono library is different from most other parts of the standard library, by using a nested namespace, std::chrono. Within that namespace, the system_clock class has a member typedef, time_point, and a function, now().

The now() function is static, so it is called as a qualified name, not by using a member access operator. Although it operates on an object, it behaves as a free function. The only difference between now() and a completely free function is that its name is qualified by a class name instead of a namespace name. Exploration 50 briefly touched on static functions. They are not used often, but this is one of those instances when such a function is useful. The now() function is declared with the static qualifier, which means the function does not need an object, the function body has no this pointer, and the usual way to call the function is with a qualified name.

A data member may be static too. (Go back to Exploration 40 for a refresher.) A member function (ordinary or static) can refer to a static data member normally, or you can use a qualified name to access the member from outside the class. One additional difference between a static member function and a free function is that a static member function can access private static members of the class. If you declare a static data member, you must also provide a definition for that member, typically in the same source file where member functions are defined. Recall that a non-static data member does not have a definition, because the instance of the data member is created when the containing object is created. Static data members are independent of any objects, so they must be defined independently too.

In Listing 68-2, the first call to d.value() calls base::value(). Without the *using* declaration in derived, the only signature for value() is value(value_type i), which doesn't match value(), and so would result in a compile error. But the *using* inner::base::value declaration injects the value name from inner::base, adding the functions value() and value(long) as additional functions overloading the name value. Thus, when the compiler looks up d.value(), it searches all three signatures to find the value() that the *using* declaration injected into derived. The second call, d.value(42L), invokes value(long). Even though the function is static, it can be called using a member access operator. The compiler ignores the object but uses the object's type as the context for looking up the name. The final call to value(2) is qualified by the class name, so it searches only the value functions in class base, finds value(long), and converts the int 2 to long.

In the most_derived class template, the base class depends on the template parameter, T. Thus, the base_type typedef is dependent. The compiler needs to know what base_type::value_type is, so the typename keyword informs the compiler that value_type is a type.

Unqualified Name Lookup

A name without a member access operator or a qualifier is *unqualified*. The precise rules for looking up an unqualified name depend on the context. For example, inside a member function, the compiler searches other members of the class and then inherited members, before searching in the class's namespace and then outer namespaces.

The rules are commonsense, and the details are germane, primarily to compiler writers who must get all the details correct. For most programmers, you can get by with common sense and a few guidelines.

- Names are looked up first in the local scope, then in outer scopes.
- In a class, names are looked up among the class members, then in ancestor classes.
- In a template, the compiler must resolve every unqualified object and type name when the template is defined, without regard to the instantiation context. Thus, it does *not* search a base class for names, if the base class depends on a template parameter.
- If a name cannot be found in the class or ancestors, or if the name is called outside of any class context, the compiler searches the immediate namespace, then outer namespaces.
- If a name is a function or operator, the compiler also searches the namespaces of the function arguments and their outer namespaces, according to the rules of argument-dependent lookup (ADL). In a template, namespaces for the template declaration and instantiation are searched.

Listing 68-3 contains several examples of unqualified name lookup.

Listing 68-3. Unqualified Name Lookup

```
#include <iostream>

namespace outer {
   namespace inner {
      struct point { int x, y; };
      inline std::ostream& operator<<(std::ostream& stream, point const& p)
      {
         stream << '(' << p.x << ", " << p.y << ')';
         return stream;
      }
   }
}

typedef int Integer;

int main()
{
   const int multiplier{2};
   for (Integer i : { 1, 2, 3}) {
      outer::inner::point p{ i, i * multiplier };
      std::cout << p << '\n';
   }
}
```

Argument-Dependent Lookup

The most interesting form of unqualified name lookup is argument-dependent lookup. As the name implies, the compiler looks up a function name in the namespaces determined by the function arguments. As a guideline, the compiler assembles the broadest, most inclusive set of classes and namespaces that it reasonably can, to maximize the search space for a name.

More precisely, if a search finds a member function, the compiler does not apply ADL, and the search stops there. Otherwise, the compiler assembles an additional set of classes and namespaces to search and combines them with the namespaces it searches for normal lookup. The compiler builds this additional set by checking the types of the function arguments. For each function argument, the class or namespace in which the argument's type is declared is added to the set. In addition, if the argument's type is a class, the ancestor classes and their namespaces are also added. If the argument is a pointer, the additional classes and namespaces are those of the base type. If you pass a function as an argument, that function's parameter types are added to the search space. When the compiler searches the additional ADL-only namespaces, it searches only for matching function names, ignoring types and variables.

If the function is a template, the additional classes and namespaces include those where the template is defined and where the template is instantiated.

Listing 68-4 shows several examples of argument-dependent lookup. The listing uses the definition of `rational` from Exploration 52, in the `numeric` namespace.

Listing 68-4. Argument-Dependent Name Lookup

```
#include <cmath>
#include <iostream>
#include "rational.hpp"

namespace data {
  template<class T>
  struct point {
    T x, y;
  };
  template<class Ch, class Tr, class T>
  std::basic_ostream<Ch, Tr>& operator<<(std::basic_ostream<Ch, Tr>& stream, point<T> const& pt)
  {
    stream << '(' << pt.x << ", " << pt.y << ')';
    return stream;
  }
  template<class T>
  T abs(point<T> const& pt) {
    using namespace std;
    return sqrt(pt.x * pt.x + pt.y * pt.y);
  }
}

namespace numeric {
   template<class T>
   rational<T> sqrt(rational<T> r)
   {
     using std::sqrt;
     return rational<T>{sqrt(static_cast<double>(r))};
   }
}

int main()
{
   using namespace std;
   data::point<numeric::rational<int>> a{ numeric::rational<int>{1, 2}, numeric::rational<int>{2, 4}
};
   std::cout << "abs(" << a << ") = " << abs(a) << '\n';
}
```

Start with `main()` and follow the name lookups.

The first name is `data`, which is looked up as an unqualified name. The compiler finds the namespace `data`, declared in the global namespace. The compiler then knows to look up `point` in the `data` namespace, and it finds the class template. Similarly, the compiler looks up `numeric` and then `rational`.

The compiler constructs `a` and adds the name to the local scope.

The compiler looks up `std` and then `cout`, which it finds, because `cout` was declared in the `<iostream>` header. Next, the compiler looks up the unqualified name, `a`, which it finds in the local scope. But then it has to look up `abs`.

The compiler searches first in the local scope and then in the global scope. The *using* directive tells the compiler to search namespace `std` too. That exhausts the possibilities for normal lookup, so the compiler must turn to argument-dependent lookup.

The compiler assembles its set of scopes to search. First it adds `data` to the namespaces to search. Because `point` is a template, the compiler also searches the namespace where the template is instantiated, which is the global namespace. It already searched there, but that's okay. Once the set is complete, the compiler searches for and finds `abs` in namespace `data`.

In order to instantiate the template `abs`, with the template argument `numeric::rational<int>`, the compiler must lookup `operator*`. It cannot find a declaration in the local scope, namespace `data`, namespace `std`, or the global namespace. Using argument-dependent lookup, it finds `operator*` in the `numeric` namespace, where `rational` is declared. It performs the same lookup for `operator+`.

In order to find `sqrt`, the compiler again uses argument-dependent lookup. When we last visited the rational class, it lacked a `sqrt` function, so Listing 68-4 provides a crude one. It converts the `rational` to a `double`, calls `sqrt`, and then converts back to `rational`. The compiler finds `sqrt` in namespace `std`.

Finally, the compiler must again apply argument-dependent lookup for `operator<<`. When the compiler compiles `operator<<` in `point`, it doesn't know about `operator<<` for `rational`, but it doesn't have to until the template is instantiated. As you can see, writing code that takes advantage of argument-dependent lookup is straightforward, if you follow simple guidelines. The next Exploration takes a closer look at the rules for resolving overloaded functions and operators. Again, you will find complicated rules that can be made simple by following some basic guidelines.

EXPLORATION 69

Overloaded Functions and Operators

Exploration 24 introduced the notion of overloaded functions. Exploration 30 continued the journey with overloaded operators. Since then, we've managed to get by with a commonsense understanding of overloading. I would be remiss if I did not delve deeper into this subject, so let's finish the story of overloading, by examining the rules of overloaded functions and operators in greater depth. (Operators and functions follow the same rules, so in this Exploration, understand that *functions* applies equally to functions and user-defined operators.)

Type Conversion

Before jumping into the deep end of the overloading pool, I need to fill in some missing pieces with respect to type conversion. Recall from Exploration 25 that the compiler promotes certain types to other types, such as `short` to `int`. It can also convert a type, such as `int`, to another type, such as `long`.

Another way to convert one type to another is with a one-argument constructor. You can think of `rational{1}` as a way to convert the `int` literal `1` to the `rational` type. When you declare a one-argument constructor, you can tell the compiler whether you want it to perform such type conversion implicitly or require an explicit type conversion. That is, if the constructor is implicit (the default), a function that declares its parameter type to be `rational` can take an integer argument, and the compiler automatically constructs a `rational` object from the `int`, as in the following:

```
rational reciprocal(rational const& r)
{
  return rational{r.denominator(), r.numerator()};
}
rational half{ reciprocal(2) };
```

To prohibit such implicit construction, use the `explicit` specifier on the constructor. That forces the user to explicitly name the type in order to invoke the constructor. For example, `std::vector` has a constructor that takes an integer as its sole argument, which initializes the vector with that many default-initialized elements. The constructor is `explicit` to avoid statements such as the following:

```
std::vector<int> v;
v = 42;
```

If the constructor were not `explicit`, the compiler would automatically construct a `vector` from the integer 42 and assign that `vector` to `v`. Because the constructor is `explicit`, the compiler balks and reports an error.

Another way to convert one type to another is with a type-conversion operator. Write such an operator with the `operator` keyword, followed by the destination type. Like a one-argument constructor, you can declare the type-conversion operator to be `explicit`. Instead of the `convert` or `as_float` functions in `rational`, you could also write type conversion operators, as follows:

```
explicit operator float() const{
    return float(numerator()) / float(denominator());}
```

One context in which the compiler automatically invokes a type-conversion operator is a loop, or if-statement, condition. Because you are using the expression in a condition, and the condition must be Boolean, the compiler considers such a use to be an explicit conversion to type `bool`. If you implement a type conversion operator for type `bool`, always use the `explicit` specifier. You will be able to test objects of your type in a condition, and you will avoid a nasty problem in which the compiler converts your type to `bool` and then promotes the `bool` to `int`. You don't really want to be able to write, for example:

```
int i;
i = std::cin;
```

In the following discussion of overload resolution, type conversion plays a major role. The compiler doesn't care how it converts one type to another, only whether it must perform a conversion, and whether the conversion is built into the language or user-defined. Constructors are equivalent to type-conversion operators.

Review of Overloaded Functions

Let's refresh the memory a bit. A function name is *overloaded* when two or more function declarations declare the same name in the same scope. C++ imposes some restrictions on when you are allowed to overload a function name.

The primary restriction is that overloaded functions must have different argument lists. This means the number of arguments must be different, or the type of at least one argument must be different.

```
void print(int value);
void print(double value);         // valid overload: different argument type
void print(int value, int width); // valid overload: different number of arguments
```

You are not allowed to define two functions in the same scope when the functions differ only in the return type.

```
void print(int);
int print(int);  // illegal
```

Member functions can also differ by the presence or absence of the `const` qualifier.

```
class demo {
   void print();
   void print() const; // valid: const qualifier is different
};
```

A member function cannot be overloaded with a static member function in the same class.

```
class demo {
   void print();
   static void print(); // illegal
};
```

A key point is that overloading occurs within a single scope. Names in one scope have no influence or impact on names in another scope. Remember that a code block is a scope (Exploration 13), a class is a scope (Exploration 40), and a namespace is a scope (Exploration 52).

Thus, member functions in a base class are in that class's scope and do not impact overloading of names in a derived class, which has its own scope, separate and distinct from the base class's scope.

When you define a function in a derived class, it hides all functions with the same name in a base class or in an outer scope, even if those functions take different arguments. This rule is a specific example of the general rule that a name in an inner scope hides names in outer scopes. Thus, any name in a derived class hides names in base classes and at namespace scope. Any name in a block hides names in outer blocks, and so on. The only way to call a hidden function from a derived class is to qualify the function name, as shown in Listing 69-1.

Listing 69-1. Qualifying a Member Function with the Base Class Name

```
#include <iostream>

class base {
public:
   void print(int x) { std::cout << "int: " << x << '\n'; }
};
class derived : public base {
public:
   void print(double x) { std::cout << "double: " << x << '\n'; }
};
int main()
{
   derived d{};
   d.print(3);           // prints double: 3
   d.print(3.0);         // prints double: 3
   d.base::print(3);     // prints int: 3
   d.base::print(3.0);   // prints int: 3
}
```

Sometimes, however, you want overloading to take into account functions in the derived class and the functions from the base class too. The solution is to inject the base class name into the derived class scope. You do this with a *using* declaration (Exploration 52). **Modify Listing 69-1 so `derived` sees both `print` functions.** Change `main` so it calls `d.print` with an `int` argument and with a `double` argument, with no qualifying names. **What output do you expect?**

____

____

Try it and compare your result with that in Listing 69-2.

Listing 69-2. Overloading Named with a using Declaration

```
#include <iostream>

class base {
public:
   void print(int x) { std::cout << "int: " << x << '\n'; }
};
```

```
class derived : public base {
public:
   void print(double x) { std::cout << "double: " << x << '\n'; }
   using base::print;
};
int main()
{
   derived d{};
   d.print(3);            // prints int: 3
   d.print(3.0);          // prints double: 3
}
```

A *using* declaration imports all the overloaded functions with that name. To see this, **add `print(long)` to the base class and a corresponding function call to `main`.** Now your example should look something like Listing 69-3.

Listing 69-3. Adding a Base Class Overload

```
#include <iostream>

class base {
public:
   void print(int x) { std::cout << "int: " << x << '\n'; }
   void print(long x) { std::cout << "long: " << x << '\n'; }
};
class derived : public base {
public:
   void print(double x) { std::cout << "double: " << x << '\n'; }
   using base::print;
};
int main()
{
   derived d{};
   d.print(3);           // prints int: 3
   d.print(3.0);         // prints double: 3
   d.print(3L);          // prints long: 3
}
```

The overload rules usually work well. You can clearly see which `print` function the compiler selects for each function call in `main`. Sometimes, however, the rules get murkier.

For example, suppose you were to add the line `d.print(3.0f);` to `main`. **What do you expect the program to print?**

____

The compiler promotes the `float 3.0f` to type `double` and calls `print(double)`, so the output is as follows:

```
double: 3
```

That was too easy. What about a `short`? **Try `d.print(short(3))`. What happens?**

____

The compiler promotes the `short` to type `int` and produces the following output:

```
int: 3
```

That was still too easy. Now try `unsigned`. **Add `d.print(3u)`. What happens?**

____

That doesn't work at all, does it? The error message probably says something about an ambiguous overload or function call. To understand what went wrong, you need a better understanding of how overloading works in C++, and that's what the rest of this Exploration is all about.

Overload Resolution

The compiler applies its normal lookup rules (Exploration 68) to find the declaration for a function name. The compiler stops searching when it finds the first occurrence of the desired name, but that scope may have multiple declarations with the same name. For a type or variable, that would be an error, but functions may have multiple, or overloaded, declarations with the same name.

After the compiler finds a declaration for the function name it is looking up, it finds all the function declarations of that name in the same scope and applies its overloading rules to choose the one declaration that it deems to match the function arguments the best. This process is called *resolving* the overloaded name.

To resolve an overload, the compiler considers the arguments and their types, the types of the function parameters in the function declarations, and type conversions and promotions that are necessary to convert the argument types to match the parameter types. Like name lookup, the detailed rules are complicated, subtle, and sometimes surprising. But if you avoid writing pathological overloads, you can usually get by with some commonsense guidelines.

Overload resolution starts after the compiler finds a declaration of the function name. The compiler collects all declarations of the same name in the same scope. This means that the compiler does not include functions of the same name from any base or ancestor classes. A *using* declaration can bring such names into the derived class scope and, thus, have them participate in overload resolution. If the function name is unqualified, the compiler looks for member and nonmember functions. A *using* directive, on the other hand, has no effect on overload resolution, because it does not alter any names in a namespace.

If the function is a constructor, and there is one argument, the compiler also considers type-conversion operators that return the desired class or a derived class.

The compiler then discards any functions with the wrong number of parameters or those for which the function arguments cannot be converted to the corresponding parameter type. When matching member functions, the compiler adds an implicit parameter, which is a pointer to the object, as though `this` were a function parameter.

Finally, the compiler ranks all the remaining functions by measuring what it needs to do to convert each argument to the corresponding parameter type, as explained in the next section. If there is a unique winner with the best rank, that compiler has successfully resolved the overload. If not, the compiler applies a few tie-breaker rules to try to select the best-ranked function. If the compiler cannot pick a single winner, it reports an ambiguity error. If it has a winner, it continues with the next compilation step, which is to check access levels for member functions. The best-ranked overload might not be accessible, but that doesn't impact how the compiler resolves the overload.

Ranking Functions

In order to rank functions, the compiler determines how it would convert each argument to the corresponding parameter type. The executive summary is that the best-ranked function is the one that requires the least work to convert all the arguments to the desired parameter types.

The compiler has several tools at its disposal to convert one type to another. Many of these you've seen earlier in the book, such as promoting arithmetic types (Exploration 25), converting a derived-class reference to a base-class reference (Exploration 39), or calling a type-conversion operator. The compiler assembles a series of conversions into an *implicit conversion sequence* (ICS). An ICS is a sequence of small conversion steps that the compiler can apply to a function-call argument with the end result of converting the argument to the type of the corresponding function parameter.

The compiler has ranking rules to determine whether one ICS is better than another. The compiler tries to find one function for which the ICS of every argument is the best (or tied for best) ICS among all overloaded names, and at least one ICS is unambiguously the best. If so, it picks that function as the best-ranked. Otherwise, if it has a set of functions that are all tied for best set of ICSes, it goes to a tie-breaker, as described in the next section. The remainder of this section discusses how the compiler ranks ICSes.

First, some terminology. An ICS may involve standard conversions or user-defined conversions. A *standard* conversion is inherent in the C++ language, such as arithmetic conversions. A *user-defined* conversion involves constructors and type conversion operators on class and enumerated types. A *standard ICS* is an ICS that contains only standard conversions. A *user-defined ICS* consists of a series of standard conversions with one user-defined conversion anywhere in the sequence. (Thus, any overload that requires two user-derived conversions in order to convert the argument to the parameter type never even gets this far, because the compiler cannot convert the argument to the parameter type and so drops that function signature from consideration.)

For example, converting `short` to `const int` is a standard ICS with two steps: promoting `short` to `int` and adding the `const` qualifier. Converting a character string literal to `std::string` is a user-defined ICS that contains a standard conversion (array of `const char` converted to pointer to `const char`), followed by a user-defined conversion (the `std::string` constructor).

One exception is that invoking a copy constructor to copy identical source and destination type or derived-class source to a base-class type is a standard conversion, not a user-defined conversion, even though the conversions invoke user-defined copy constructors.

The compiler has to pick the best ICS of the functions that remain under consideration. As part of this determination, it must be able to compare standard conversions within an ICS. A standard conversion falls into one of three categories. In order from best to worst, the categories are exact match, promotion, and other conversion.

An *exact match* is when the argument type is the same as the parameter type. Examples of exact match conversions are

- Changing only the qualification, e.g., the argument is type `int` and the parameter is `const int` (but not pointer to `const` or reference to `const`)
- Converting an array to a pointer (Exploration 59), e.g., `char[10]` to `char*`
- Converting an lvalue to an rvalue, e.g., `int&` to `int`

A *promotion* (Exploration 25) is an implicit conversion from a smaller arithmetic type (such as `short`) to a larger type (such as `int`). The compiler considers promotion better than conversion, because a promotion does not lose any information, but a conversion might.

All other implicit type conversions—for example, arithmetic conversions that discard information (such as `long` to `int`) and derived-class pointers to base-class pointers—fall into the final category of miscellaneous conversions.

The category of a sequence is the category of the worst conversion step in the sequence. For example, converting `short` to `const int` involves an *exact match* (`const`) and a *promotion* (`short` to `int`), so the category for the ICS as a whole is *promotion*.

If one argument is an implicit object argument (for member function calls), the compiler compares any conversions needed for it too.

Now that you know how the compiler orders standard conversions by category, you can see how it uses this information to compare ICSes. The compiler applies the following rules to determine which of two ICSes is better:

- A standard ICS is better than a user-defined ICS.
- An ICS with a better category is better than an ICS with a worse category.
- An ICS that is a proper subset of another ICS is better.

- A user-defined ICS, ICS1, is better than another user-defined ICS, ICS2, if they have the same user conversion and the second standard conversion in ICS1 is better than the second standard conversion of ICS2.
- Less restrictive types are better than more restrictive ones. This means an ICS with target type T1 is better than an ICS with target type T2, if T1 and T2 have the same base type, but T2 is const and T1 is not.
- A standard conversion sequence ICS1 is better than ICS2, if they have the same rank, but
    - ICS1 converts a pointer to `bool`, or
    - ICS1 and ICS2 convert pointers to classes related by inheritance, and ICS1 is a “smaller” conversion. A smaller conversion is one that hops over fewer intermediate base classes. For example, if `A` derives from `B` and `B` from `C`, then converting `B*` to `C*` is better than converting `A*` to `C*`, and converting `C*` to `void*` is better than `A*` to `void*`.

List Initialization

One complication is the possibility of a function argument that has no type because it is not an expression. Instead, the argument is a curly-brace-enclosed list of values, such as the brace-enclosed list that is used for universal initialization. The compiler has some special rules for determining the conversion sequence of a list.

If the parameter type is a class with a constructor that takes a single argument of type `std::initializer_list<T>`, and every member of the brace-enclosed list can be converted to `T`, the compiler treats the argument as a user-defined conversion to `std::initializer_list<T>`. All the container classes, for example, have such a constructor.

Otherwise, the compiler tries to find a constructor for the parameter type such that each element of the brace-enclosed list is an argument to the constructor. If it succeeds, the compiler considers the list a user-defined conversion to the parameter type. Note that another user-defined conversion sequence is allowed for each constructor argument.

The compiler considers `std::initializer_list` initialization better than the other constructor list initialization. That’s why `std::string{42, 'x'}` does not invoke the `std::string(42, 'x')` constructor: the compiler prefers treating `{42, 'x'}` as `std::initializer_list`, which results in a string with two characters, one with code point 42 and the letter *x*, and not the constructor that creates a string with 42 repetitions of the letter *x*.

If the parameter type is not a class, and the brace-enclosed list contains a single element, the compiler unwraps the value from the curly braces and applies the normal ICS that results from the enclosed value.

Tie-Breakers

If the compiler cannot find one function that unambiguously ranks higher than the others, it applies a few final rules to try to pick a winner. The compiler checks the following rules in order. If one rule yields a winner, the compiler stops at that point and uses the winning function. Otherwise, it continues with the next tie-breaker

- Although return type is not considered part of overload resolution, if the overloaded function call is used in a user-defined initialization, the function’s return type that invokes a better standard conversion sequence wins.
- A non-template function beats a function template.
- A more-specialized function template beats a less-specialized function template. (A reference or pointer template parameter is more specialized than a non-reference or non-pointer parameter. A `const` parameter is more specialized than non-`const`.)
- Otherwise, the compiler reports an ambiguity error.

Listing 69-4 shows some examples of overloading and how C++ ranks functions.

Listing 69-4. Ranking Functions for Overload Resolution

```
#include <iostream>
#include <string>

void print(std::string const& str) { std::cout << str; }
void print(int x)                  { std::cout << "int: " << x; }
void print(double x)               { std::cout << "double: " << x; }

class base {
public:
  void print(std::string const& str) const { ::print(str); ::print("\n"); }
  void print(std::string const& s1, std::string const& s2)
  {
    print(s1); print(s2);
  }
};

class convert : public base {
public:
  convert()              { print("convert()"); }
  convert(double)        { print("convert(double)"); }
  operator int() const   { print("convert::operator int()"); return 42; }
  operator float() const { print("convert::operator float()"); return 3.14159f; }
};

class demo : public base {
public:
  demo(int)      { print("demo(int)"); }
  demo(long)     { print("demo(long)"); }
  demo(convert)  { print("demo(convert)"); }
  demo(int, int) { print("demo(int, int)"); }
};

class other {
public:
  other()        { std::cout << "other::other()\n"; }
  other(int,int) { std::cout << "other::other(int, int)\n"; }
  operator convert() const
  {
    std::cout << "other::operator convert()\n"; return convert();
  }
};

int operator+(demo const&, demo const&)
{
  print("operator+(demo,demo)\n"); return 42;
}

```

```
int operator+(int, demo const&) { print("operator+(int,demo)\n"); return 42; }

int main()
{
  other x{};
  demo d{x};
  3L + d;
  short s{2};
  d + s;
}
```

What output do you expect from the program in Listing 69-4?

Most of the time, commonsense rules help you understand how C++ resolves overloading. Sometimes, however, you find the compiler reporting an ambiguity when you did not expect any. Other times, the compiler cannot resolve an overload when you expected it to succeed. The really bad cases are when you make a mistake and the compiler is able to find a unique function, but one that is different from the one you expect. Your tests fail, but when reading the code, you look in the wrong place, because you expect the compiler to complain about bad code.

Sometimes, your compiler helps you by identifying the functions that are tied for best rank. Sometimes, however, you might have to sit down with the rules and go over them carefully to figure out why the compiler isn't happy. To help you prepare for that day, Listing 69-5 presents some overloading errors. **See if you can find and fix the problems.**

Listing 69-5. Fix the Overloading Errors

```
#include <iostream>
#include <string>

void easy(long) {}
void easy(double) {}
void call_easy() {
   easy(42);
}

void pointer(double*) {}
void pointer(void*) {}
const int zero = 0;
void call_pointer() {
   pointer(&zero);
}

int add(int a) { return a; }
int add(int a, int b) { return a + b; }
int add(int a, int b, int c) { return a + b + c; }
int add(int a, int b, int c, int d) { return a + b + c + d; }
int add(int a, int b, int c, int d, int e) { return a + b + c + d + e; }
void call_add() {
   add(1, 2, 3L, 4.0);
}

void ref(int const&) {}
void ref(int) {}
void call_ref() {
   int x;
   ref(x);
}

class base {};
class derived : public base {};
class sibling : public base {};
class most_derived : public derived {};

void tree(derived&, sibling&) {}
void tree(most_derived&, base&) {}
void call_tree() {
   sibling s;
   most_derived md;
   tree(md, s);
}
```

The argument to `easy()` is an `int`, but the overloads are for `long` and `double`. Both conversions have conversion rank, and neither one is better than the other, so the compiler issues an ambiguity error.

The problem with `pointer()` is that neither overload is viable. If `zero` were not `const`, the conversion to `void*` would be the sole viable candidate.

The `add()` function has all `int` parameters, but one argument is `long` and another is `double`. No problem, the compiler can convert `long` to `int` and `double` to `int`. You may not like the results, but it is able to do it, so it does. In other words, the problem here is that the compiler does not have a problem with this function. This isn't really an overloading problem, but you may not see it that way if you run into this problem at work.

Do you see the missing `&` in the second `ref()` function? The compiler considers both `ref()` functions to be equally good. If you declare the second to be `ref(int&)`, it becomes the best viable candidate. The exact reason is that the type of `x` is `int&`, not `int`, that is, `x` is an `int` lvalue, an object that the program can modify. The subtle distinction has not been important before now, but with respect to overloading, the difference is crucial. The conversion from an lvalue to an rvalue has rank exact match, but it is a conversion step. The conversion from `int&` to `int const&` also has exact match. Faced with two candidates with one exact match conversion each, the compiler cannot decide which one is better. Changing `int` to `int&` removes the conversion step, and that function becomes the unambiguous best.

Both `tree()` functions require one conversion from derived-class reference to base-class reference, so the compiler cannot decide which one is better. The first call to `tree` requires a conversion of the first argument from `most_derived&` to `derived&`. The second call requires a conversion of the second argument from `sibling&` to `base&`.

Remember that the purpose of overloading is to allow a single logical operation across a variety of types, or to allow a single logical operation (such as constructing a string) to be invoked in a variety of ways. These rules will help guide you to make good choices when you decide to overload a function.

■ **Tip** When you write overloaded functions, you should make sure that every implementation of a particular function name has the same logical behavior. For example, when you use an output operator, `cout << x`, you just let the compiler pick the correct overload for `operator<<`, and you don't have to concern yourself with the detailed rules as laid out in this Exploration. All the rules apply, but the standard declares a reasonable set of overloads that work with the built-in types and key library types, such as `std::string`.

Default Arguments

Now that you think overloading is so frightfully complicated that you never want to overload a function, I will add yet another complexity. C++ lets you define a default argument for a parameter, which lets a function call omit the corresponding argument. You can define default arguments for any number of parameters, provided you omit the right-most arguments and don't skip any. You can provide default arguments for every parameter if you wish. Default arguments are often easy to understand. Read Listing 69-6 for an example.

Listing 69-6. Default Arguments

```
#include <iostream>

int add(int x = 0, int y = 0)
{
  return x + y;
}

int main()
{
  std::cout << add() << '\n';
  std::cout << add(5) << '\n';
  std::cout << add(32, add(4, add(6))) << '\n';
}
```

What does the program in Listing 69-6 print?

____

____

____

It's not hard to predict the results, which are shown in the following output:

```
0
5
42
```

Default arguments offer a shortcut, in lieu of overloading. For example, instead of writing several constructors for the `rational` type, you can get by with one constructor and default arguments, as follows:

```
template<class T> class rational {
public:
  rational(T const& num = T{0}, T const& den = T{1})
  : numerator_{num}, denominator_{den}
  {
    reduce();
  }
  ...omitted for brevity...
};
```

Our definition of a default constructor must change somewhat. Instead of being a constructor that declares no parameters, a default constructor is one that you can call with no arguments. This `rational` constructor meets that requirement.

As you may have guessed, default arguments complicate overload resolution. When the compiler searches for overloaded functions, it checks every argument that explicitly appears in the function call but does not check default argument types against their corresponding parameter types. As a result, you can run into ambiguous situations more easily with default arguments. For example, suppose you added the example `rational` constructor to the existing class template without deleting the old constructors. The following definitions would both result in ambiguity errors:

```
rational<int> zero{};
rational<int> one{1};
```

Default arguments have their uses, but overloading usually gives you more control. For example, by overloading the rational constructors, we avoid calling `reduce()` when we know the denominator is 1. Using inline functions, one overloaded function can call another, which often eliminates the need for default arguments completely. If you are unsure whether to use default arguments or overloading, I recommend overloading.

Although you may not believe me, my intention was not to scare you away from overloading functions. Rarely will you have to delve into the subtleties of overloading. Most of the time, you can rely on common sense. But sometimes, the compiler disagrees with your common sense. Knowing the compiler's rules can help you escape from a jam when the compiler complains about an ambiguous overload or other problems.

The next Exploration visits another aspect of C++ programming for which the rules can be complicated and scary: metaprogramming, or writing programs that run at compile time.

EXPLORATION 70

Metaprogramming

Metaprogramming is the act of writing programs that run during the compilation of an ordinary program. Most metaprograms use templates to operate on types, but you can also write constexpr functions to compute values at compile time. Metaprogramming requires a different style than ordinary programming, and you will find yourself calling upon almost all the programming techniques you have learned so far.

Use constexpr for Compile-Time Values

In C++ 03, the only way to work with numeric values at compile time was to use templates and metaprogramming. In C++ 11, a better way is to use constexpr functions. To compare these two styles of programming, consult Listing 51-6, which presents the power10 function to compute powers to 10 at compile time. The equivalent function can also be implemented with templates, as shown in Listing 70-1, which was how the first edition of this book computed a power of 10 at compile time.

Listing 70-1. Computing a Power of 10 at Compile Time with Templates

```
template<int N>
struct power10; // forward declaration

template<int N>
struct square
{
  // A metaprogramming function to square a value.
  enum t { value = N * N };
};

template<int N, bool Even>
struct power10_aux
{
  // Primary template is for odd N (Even is false)
  enum t { value = 10 * power10<N - 1>::value };
};

template<int N>
struct power10_aux<N, true>
{
  // Specialization when N is even: square the value
  enum t { value = square<power10<N / 2>::value>::value };
};
```

```
template<int N>
struct power10
{
  enum t { value = power10_aux<N, N % 2 == 0>::value };
};

template<>
struct power10<0>
{
  enum t { value = 1 };
};
```

The metaprogram, like almost all template metaprograms, uses template specialization and partial specialization. This metaprogram uses only techniques that you have learned over the course of this book, but it is daunting to read and hard to follow. Metaprograms that compute values can often be written much more easily with `constexpr` functions, as you saw in Listing 51-6.

Metaprogramming with `constexpr` functions is slightly harder than writing ordinary functions but much easier than writing template metaprograms. Because the only executable statement you can place in a `constexpr` function is a `return` statement, you must learn to think in terms functional programming, using helper functions and recursion. You are limited to built-in types and custom types that can be constructed with `constexpr` constructors. But in spite of these limitations, you can achieve a lot, such as precomputing constants, or extending the language with user-defined literals.

User-defined literals were introduced in Exploration 25. They combine nicely with `constexpr` to extend the language at compile time. For example, if you often work with bit patterns, you probably write constants in hexadecimal, and you mentally map base 2 to and from base 16. It would be simpler if you could write literals directly in base 2. For example, suppose you want to write the following:

```
enum { answer = 00101010_binary }; // answer == 0x2a == 42
```

An enumerated literal must be a compile-time constant, so the `_binary` literal operator must be a `constexpr` function. The function interprets the series of digits as a binary number, as shown in Listing 70-2.

Listing 70-2. Implementing the `_binary` User-Defined Literal Operator

```
/// Compute one bit of a binary integer.
/// Compute a new result by taking the right-most decimal digit from @p digits,
/// and if it is 1, shifting the 1 left by @p shift places and bitwise ORing
/// the value with @p result. Ignore digits other than 0 or 1. Recurse
/// to compute the remaining result.
/// @param digits The user-supplied decimal digits, should use only 0 and 1
/// @param result The numeric result so far
/// @param shift The number of places to shift the right-most digit in @p digits
/// @return if @p digits is zero, return @p result; otherwise return the result
/// of converting @p digits to binary
constexpr unsigned long long binary_helper(unsigned long long digits,
    unsigned long long result, int shift)
{
    return digits == 0 ?
        result :
        binary_helper(digits / 10,
            result | ((digits % 10 == 1) << shift),
            digits % 10 < 2 ? shift + 1 : shift);
}
```

```
constexpr unsigned long long operator"" _binary(unsigned long long digits)
{
   return binary_helper(digits, 0ULL, 0);
}
```

Because the body of a `constexpr` function must contain only a `return` statement, conditionals require the use of the conditional operator, which is harder to read than an `if` statement but not too difficult. The `digits` string is treated as though it were in base 2, ignoring any digits other than 1 or 0. (Error-handling is one difficulty with `constexpr` functions. There is no standard way for a `constexpr` function such as this `_binary` operator to report a compile-time error to the user. Later, you will learn a different way to write this operator that will allow some error-checking.) This particular implementation of the `_binary` literal is limited to binary numbers that look like `unsigned long long` decimal values, so you cannot express the full range of `unsigned long long` base 2 values. The next section presents a technique that can eliminate this restriction.

Variable-Length Template Argument Lists

You can define a template that takes any number of template arguments (called a *variadic template*). This ability gives rise to a number of programming tricks. Most of the tricks are used by library authors, but that doesn't mean others can't join in. To declare a template parameter that can accept any number of arguments, use an ellipsis after the `class` or `typename` keyword for a type template parameter, or after the type in a value template parameter. Such a parameter is called a *parameter pack*. Following are some simple examples:

```
template<class... Ts> struct List {};
template<int... Ns> struct Numbers {};
```

Instantiate the template with any number of template arguments:

```
typedef List<int> Int_type;
typedef List<char, unsigned char, signed char> Char_types;
typedef Numbers<1, 2, 3> One_two_three;
```

You can also declare a function parameter pack, so the function can take any number of arguments of any type, such as the following:

```
template<class... Types>
void list(Types... args);
```

When you call the function, the parameter pack contains the type of each function argument. In the following example, the compiler implicitly determines that the Types template argument is `<int, char, std::string>`.

```
list(1, 'x', std::string("yz"));
```

The `sizeof...` operator returns the number of elements in the parameter pack. You can, for example, define a `Size` template to compute the number of arguments in a parameter pack, as shown in the following:

```
template<class... Ts>
struct Size { constexpr static std::size_t value = sizeof...(Ts); };
static_assert(Size<int, char, long>::value == 3, "oops");
```

The static_assert declaration checks a compile-time Boolean expression and causes a compiler error if the condition is false. The string literal argument is the text of the message, and you will usually want something more informative than “oops.” Using static_assert is a great way to verify that a metaprogram does what you intend. It is useful in ordinary programs too. The more problems you can detect at compile time, the better.

To use a parameter pack, you typically expand it with a pattern followed by an ellipsis. The pattern can be the parameter name, a type that uses the parameter, an expression that uses the function parameter pack, etc.

Listing 70-3 shows a print() function that takes a stream followed by any number of arguments of any type. It prints each value by expanding the parameter pack. The std::forward() function forwards a value to a function without altering or copying it (called “perfect forwarding”). The compiler expands the pack expression std::forward<Types>(rest)... into std::forward(r) for each argument r in rest. By passing rvalue references everywhere and using std::forward(), the print() function can pass references to its arguments with a minimum of overhead. Notice that nowhere is there a test for the size of the parameter pack. The pack is expanded at compile time, and an overloaded function ends the expansion when the pack is empty.

Listing 70-3. Using a Function Parameter Pack to Print Arbitrary Values

```
#include <iostream>
#include <utility>

// Forward declaration.
template<class... Types>
void print(std::ostream& stream, Types&&...);

// Print the first value in the list, then recursively
// call print() to print the rest of the list.
template<class T, class... Types>
void print_split(std::ostream& stream, T&& head, Types&& ... rest)
{
   stream << head << ' ';
   print(stream, std::forward<Types>(rest)...);
}

// End recursion when there are no more values to print.
void print_split(std::ostream&)
{}

// Print an arbitrary list of values to a stream.
template<class... Types>
void print(std::ostream& stream, Types&&... args)
{
   print_split(stream, std::forward<Types>(args)...);
}

int main()
{
   print(std::cout, 42, 'x', "hello", 3.14159, 0, '\n');
}
```

You can use a function parameter pack with a user-defined literal too. Instead of taking an unsigned long long argument, you can implement the operator as a template function. The template arguments are the characters that make up the literal. Thus if the source code contains 00101010_binary, the template arguments are <'0', '0', '1', '0', '1', '0', '1', '0'>. Because you don’t know beforehand how many template arguments to expect, you have to use a parameter pack, as shown in Listing 70-4.

Listing 70-4. Using a Function Parameter Pack to Implement the `_binary` Operator

```
/// Extract one bit from a bit string and then recurse.
template<char Head, char... Rest>
struct binary_helper
{
   constexpr unsigned long long operator()(unsigned long long result) const;
};

/// Teminate recursion when interpreting a bit string.
template<char Head>
struct binary_helper<Head>
{
   constexpr unsigned long long operator()(unsigned long long result) const;
};

template<char Head, char... Rest>
constexpr unsigned long long
binary_helper<Head, Rest...>::operator()(unsigned long long result)
const
{
   static_assert(Head == '0' or Head == '1', "_binary contains only 0 or 1");
   return binary_helper<Rest...>{}(result << 1 | (Head - '0'));
}

template<char Head>
constexpr unsigned long long
binary_helper<Head>::operator()(unsigned long long result)
const
{
   static_assert(Head == '0' or Head == '1', "_binary contains only 0 or 1");
   return result << 1 | (Head - '0');
}

template<char... Digits>
constexpr unsigned long long operator"" _binary()
{
   return binary_helper<Digits...>{}(0ULL);
}
```

By now, you should recognize the familiar pattern. A helper function strips the first template argument and recursively calls itself with the remaining parameter pack. Partial specialization terminates the recursion. Functions cannot be partially specialized, so Listing 70-3 partially specializes the `binary_helper` functor.

As a side benefit, using a template gives you the opportunity for additional error-checking with `static_assert`. Because `static_assert` does not produce any executable code, you can have a `static_assert` in a constexpr function.

What does your compiler do when you try the following declaration after defining `_binary`?

constexpr int two = 2_binary;

You should get a message about the `static_assert` failing.

Types as Values

Metaprogramming with values is made easier with constexpr functions, but much metaprogramming involves types, which requires an entirely different viewpoint. When metaprogramming with types, a type takes on the role of a value. There is no way to define a variable, only template arguments, so you devise templates that declare the template parameters you need to store type information. A "function" in a metaprogram (sometimes called a *metafunction*) is just another template, so its arguments are template arguments.

For example, the standard library contains the metafunction is_same (defined in <type_traits>). This template takes two template arguments and yields a type as its result. The metafunctions in the standard library return a result with class members. If the result is a type, the member typedef is called type. The type member for a predicate such as is_same is a metaprogramming Boolean value. If the two argument types are the same type, the result is std::true_type (also defined in <type_traits>). If the arguments types are different, the result is std::false_type.

Because true_type and false_type are themselves metaprogramming types, they also have type member typedefs. The value of true_type::type is true_type; ditto for false_type. Sometimes a metaprogram has to treat a metaprogramming value as an actual value. Thus, metaprogramming types that represent values have a static data member named value. As you may expect, true_type::value is true and false_type::value is false.

How would you write **is_same**? You have to declare the member typedef type to std::true_type or std::false_type, depending on the template arguments. An easy way to do this, and to obtain the convenience value static data member at the same time, is to derive is_same from true_type or false_type, depending on the template arguments. This is a straightforward implementation of partial specialization, as you can see in Listing 70-5.

Listing 70-5. Implementing the is_same Metafunction

```
template<class T, class U>
struct is_same : std::false_type {};

template<class T>
struct is_same<T, T> : std::true_type {};
```

Let's write another metafunction, one that is not in the standard library. This one is called promote. It takes a single template argument and yields int if the template argument is bool, short, char, or variations and yields the argument itself otherwise. In other words, it implements a simplified subset of the C++ rules for integer promotion. How would you write **promote**? This time, the result is pure type, so there is no value member. The simplest way is the most direct. Listing 70-6 shows one possibility.

Listing 70-6. One Implementation of the promote Metafunction

```
template<class T> struct promote         { typedef T type; };
template<> struct promote<bool>          { typedef int type; };
template<> struct promote<char>          { typedef int type; };
template<> struct promote<signed char>    { typedef int type; };
template<> struct promote<unsigned char>  { typedef int type; };
template<> struct promote<short>         { typedef int type; };
template<> struct promote<unsigned short> { typedef int type; };
```

Another way to implement promote is to use template parameter packs. Suppose you have a metafunction, is_member, which tests its first argument to determine whether it appears in the parameter pack formed by its remaining arguments. That is, is_member<int, char> is false_type, and is_member<int, short, int, long> yields true_type. Given is_member, how would you implement promote? Listing 70-7 shows one way, using partial specialization on the result of is_member.

Listing 70-7. Another Implementation of the `promote` Metafunction

```
// Primary template when IsMember=std::true_type.
template<class IsMember, class T>
struct get_member {
   typedef T type;
};

template<class T>
struct get_member<std::false_type, T>
{
   typedef int type;
};

template<class T>
struct promote {
    typedef typename get_member<typename is_member<T,
        bool, unsigned char, signed char, char, unsigned short, short>::type, T>::type type;
};
```

Remember that `typename` is required when naming a type that depends on a template parameter. The `type` member of a metafunction certainly qualifies as a dependent type name. This implementation uses partial specialization to determine the result from `is_member`. Using `is_member` to implement `promote` might seem to be more complicated, but if the list of types is long, or is likely to grow as an application evolves, the `is_member` approach seems more inviting. Although using `is_member` is easy, writing it is not so easy. Remember how Listing 70-4 splits off the head of the function pack? Use the same technique to split the parameter pack, that is, write a helper class that has a `Head` template parameter and a `Rest` template parameter pack. Listing 70-8 shows one way to implement `is_member`.

Listing 70-8. Implementing the `is_member` Metafunction

```
template<class Check, class... Args> struct is_member;

// Helper metafunction to separate Args into Head, Rest
template<class Check, class Head, class... Rest>
struct is_member_helper :
    std::conditional<std::is_same<Check, Head>::value,
        std::true_type,
        is_member<Check, Rest...>>::type
{};

/// Test whether Check is the same type as a type in Args.
template<class Check, class... Args>
struct is_member : is_member_helper<Check, Args...> {};

// Partial specialization for empty Args
template<class Check>
struct is_member<Check> : std::false_type {};
```

Instead of writing a custom metafunction that specializes on `std::false_type`, Listing 70-8 uses a standard metafunction, `std::conditional`. It is usually better to use the standard library whenever possible, and you can rewrite Listing 70-7 to use `std::conditional`. To help you understand this important metafunction, the next section discusses `std::conditional` in depth.

Conditional Types

One key aspect of metaprogramming is making decisions at compile time. To do that, you need a conditional operator. The standard library offers two styles of conditionals in the `<type_traits>` header.

To test a condition, use `std::conditional<Condition, IfTrue, IfFalse>::type`. The `Condition` is a `bool` value, and `IfType` and `IfFalse` are types. The `type` member is a typedef for `IfTrue` if `Condition` is true and is a typedef for `IfFalse` if `Condition` is false.

Try writing your own implementation of **`std::conditional`**. Your standard library may be different but won't be to terribly different from my solution in Listing 70-9.

Listing 70-9. One Way to Implement `std::conditional`

```
template<bool Condition, class IfTrue, class IfFalse>
struct conditional
{
    typedef IfFalse type;
};

template<class IfTrue, class IfFalse>
struct conditional<true, IfTrue, IfFalse>
{
   typedef IfTrue type;
};
```

Another way to look at `std::conditional` is to consider it an array of two types, indexed by a `bool` value. What about an array of types indexed by an integer? The standard library doesn't have such a template, but you can write one. Use a template parameter pack and an integer selector. If the selector is invalid, do not define the `type` member typedef. For example, `choice<2, int, long, char, float, double>::type` would be `char`, and `choice<2, int, long>` would not declare a `type` member. Try writing **`choice`**. Again, you will probably want two mutually recursive classes. One class strips the first template parameter from the parameter pack and decrements the index. Template specialization terminates the recursion. Compare your solution with mine in Listing 70-10.

Listing 70-10. Implementing an Integer-Keyed Type Choice

```
#include <cstddef>
#include <type_traits>

// forward declaration
template<std::size_t, class...>
struct choice;

// Default: subtract one, drop the head of the list, and recurse.
template<std::size_t N, class T, class... Types>
struct choice_split {
    typedef typename choice<N-1, Types...>::type type;
};

// Index 0: pick the first type in the list.
template<class T, class... Ts>
struct choice_split<0, T, Ts...> {
    typedef T type;
};
```

```
// Define type member as the N-th type in Types.
template<std::size_t N, class... Types>
struct choice {
    typedef typename choice_split<N, Types...>::type type;
};

// N is out of bounds
template<std::size_t N>
struct choice<N> {};

// Tests

static_assert(std::is_same<int,
  typename choice<0, int, long, char>::type>::value, "error in choice<0>");
static_assert(std::is_same<long,
  typename choice<1, int, long, char>::type>::value, "error in choice<1>");
static_assert(std::is_same<char,
  typename choice<2, int, long, char>::type>::value, "error in choice<2>");
```

Use the new choice template to choose one option from among many. On one project, I defined three styles of iterators for different trade-offs of safety and performance. The fast iterator worked as fast as possible, with no safety checks. The safe iterator would check just enough to avoid undefined behavior. The pedantic iterator was used for debugging and checked everything possible, with no regard for speed. I could pick which iterator style I wanted by defining ITERATOR_TYPE as 0, 1, or 2, for example:

```
typedef typename choice<ITERATOR_TYPE,
    pedantic_iterator, safe_iterator, fast_iterator>::type iterator;
```

Checking Traits

A variation on std::conditional is std::enable_if. Like conditional, enable_if uses a bool template argument to choose a type that it uses to declare its type member typedef. The difference is that if the condition is false, enable_if does not define any type member at all (like choice when the index is out of bounds). Use enable_if to enable or disable function signatures or entire classes.

A silly, but simple, example is to define a minus function template. It takes a signed numeric argument and returns the arithmetic negation. But what if the argument is not signed? By using enable_if, you can ensure that the function template is defined only for signed types. Use std::numeric_limits (refer to Exploration 25, if you need a reminder about numeric_limits) to determine whether a type is signed. Use enable_if to enable the function only when is_signed is true, as shown in Listing 70-11.

Listing 70-11. Implementing an Integer-Keyed Type Choice

```
#include <limits>
#include <type_traits>

template<class T>
typename std::enable_if<std::numeric_limits<T>::is_signed, T>::type
minus(T const& x)
{
   return -x;
}
```

The use of enable_if is intimidating, so let's take it one step at a time. The minus() function template can take any argument type, T. The numeric_limits traits are tested to determine whether T is signed. If so, enable_if's type member will be a typedef for T, which is exactly what we want the return type to be. But if is_signed is false, enable_if does not declare any type member, and the compiler will issue an error, thereby preventing the user from calling minus() incorrectly.

You can also use enable_if to enable or disable entire classes. For example, suppose you want to restrict rational's template argument to integral types. Add a template argument and supply a default argument that uses enable_if to test whether the first template argument is an integer. If the enable_if fails, the compiler will reject the entire template. Listing 70-12 shows a skeleton of the rational class, using enable_if.

Listing 70-12. Specializing the rational Class Using enable_if

```
#include <limits>
#include <type_traits>

template<class T, class Enable = typename std::enable_if<std::numeric_limits<T>::is_integer,
T>::type>
class rational {
public:
   ... normal class definition here ...
};

rational<long> okay{1, 2};
rational<float> problem{1, 2};
```

Another enhancement to rational is to limit the floating-point conversion functions to types that are truly floating-point. Use enable_if the same way you did minus() to write the convert function (Exploration 48) or an explicit type-conversion operator (Exploration 68) as a member of rational, as follows:

```
template<class T>
operator typename std::enable_if<std::is_floating_point<T>::value, T>::type()
{
    return static_cast<T>(numerator()) / static_cast<T>(denominator());
}
```

The code is hard to read, so creating an intermediate type is helpful.

```
template<class T>
using EnableIfFloat =
    typename std::enable_if<std::is_floating_point<T>::value, T>::type;

template<class T>
operator typename EnableIfFloat<T>::type ()
{
    return static_cast<T>(numerator()) / static_cast<T>(denominator());
}
```

The enable_if class template is a powerful tool to enforce compile-time rules. It is even more powerful if you don't use it to stop compilation but to enable the compiler to continue. The following section discusses this programming technique.

Substitution Failure Is Not An Error (SFINAE)

A programming technique introduced by Daveed Vandevoorde goes by the unwieldy name of SFINAE (pronounced ess-finn-ee), for *Substitution Failure Is Not An Error*. Briefly, if the compiler attempts to instantiate an invalid template function, the compiler does not consider it an error but merely discards that instantiation from consideration when resolving overloads. Such a simple concept turns out to be extremely useful.

Not all uses of `enable_if` are to cause compile-time errors. You can also use `enable_if` with SFINAE to affect overload resolution. For example, suppose you are writing data in some data encoding, such as ASN.1 BER, XDR, JSON, etc. The details of the encoding are unimportant for this exercise. What matters is that we want to treat all integers the same, and all floating-point numbers the same, but treat integers differently from floating-point numbers. That is, we want to use templates to reduce the amount of repetitive coding, but we want distinct implementations for certain types. We cannot partially specialize functions, so we must use overloading.

The problem is how to declare three template functions, all with the name `encode`, such that one is a template function for any integer, another for any floating-point type, and another for strings.

One approach is to declare overloads for the largest integer type and largest floating-point type. The compiler will convert the actual types to the larger types. This is simple to implement but incurs a runtime cost, which can be significant in some environments. We need a better solution.

Using `enable_if`, you can declare overloaded `encode` functions but enable one function only when `is_integral` is true, another for floating-point types, and so on. The goal is not to disable the `encode()` function but to guide the compiler's resolution of overloading.

The `<type_traits>` header has several introspection traits. Every type is categorized as class, enumeration, integer, floating-point, and so on. Just to show how they work, I will use `std::is_integral` instead of `numeric_limits` to test for integral types. This book is not about binary data encoding, so the guts of this example will write text to a stream, but it will serve to illustrate how to use `enable_if`.

The normal rules of overloading still apply. That is, different functions must have different arguments. So using `enable_if` for the return type doesn't help. This time, `enable_if` will be used as another argument to `encode`, but with a default value that hides it from the caller. (Note that using for the primary argument to the function doesn't work, because it breaks the compiler's ability to deduce the template type from the function's argument type.) Specifically, the `enable_if` argument is made into a pointer type, with `nullptr` as the default value, to ensure that there is no extra code constructing or passing this additional argument. With inlining, the compiler can even optimize away the extra argument, so there is no runtime penalty. Listing 70-13 shows one way to solve this problem.

Listing 70-13. Using `enable_if` to Direct Overload Resolution

```
#include <iostream>
#include <type_traits>

template<class T>
void encode(std::ostream& stream, T const& int_value,
   typename std::enable_if<std::is_integral<T>::value, T>::type* = nullptr)
{
   // All integer types end up here.
   stream << "int: " << int_value << '\n';
}

template<class T>
void encode(std::ostream& stream, T const& enum_value,
   typename std::enable_if<std::is_enum<T>::value>::type* = nullptr)
```

```
{
   // All enumerated types end up here.
   // Record the underlying integer value.
   stream << "enum: " <<
      static_cast<typename std::underlying_type<T>::type>(enum_value) << '\n';
}

template<class T>
void encode(std::ostream& stream, T const& float_value,
   typename std::enable_if<std::is_floating_point<T>::value>::type* = nullptr)
{
   // All floating-point types end up here.
   stream << "float: " << float_value << '\n';
}

// enable_if forms cooperate with normal overloading
void encode(std::ostream& stream, std::string const& string_value)
{
   stream << "str: " << string_value << '\n';
}

int main()
{
   encode(std::cout, 1);
   enum class color { red, green, blue };
   encode(std::cout, color::blue);
   encode(std::cout, 3.0);
   encode(std::cout, std::string("string"));
}
```

That concludes your exploration of C++ 11. The next and final Exploration is a capstone project to incorporate everything you have learned. I hope that you have enjoyed your journey and that you will plan many more excursions to complete your understanding and mastery of this language.

EXPLORATION 71

Project 4: Calculator

Now is the time to apply everything you have learned in this book, by writing a simple textual calculator. If you type `1 + 2`, for example, the calculator prints `3`. This project can be as complicated as you wish or dare to make it. I recommend starting small and adding capability slowly and incrementally.

1. Start with a simple parser to read numbers and operators. If you are familiar with a parser generator, such as Bison or Antlr, go ahead and use it. If you are feeling adventurous, try learning about Spirit, which is part of the Boost project. Spirit makes use of C++ operator overloading to implement a BNF-like syntax for writing a parser in C++ without requiring additional tools. If you don't want to involve other tools or libraries, I recommend a simple LISP-like syntax, so you don't spend all your time on the parser. The code on this book's web site implements a simple, recursive-descent parser. Implement the basic arithmetic operators first: `+`, `-`, `*`, and `/`. Use `double` for all numbers. Do something helpful when dividing by zero.
2. Then add variables and the `=` operator. Initialize the calculator with some useful constants, such as `pi`.
3. The big leap forward is not to evaluate every expression when it is typed but to create a parse tree. This requires some work on the parser, not to mention the addition of the parse-tree classes, that is, classes to represent expressions, variables, and values.
4. Given variables and parse trees, it is a smaller step to define functions and call user-defined functions.
5. Finally, add the ability to save functions to a file, and load them from a file. Now you can create libraries of useful functions.
6. If you are truly ambitious, try supporting multiple types. Use the `pimpl` idiom (Exploration 63) to define a `number` class and a `number_impl` class. Let the calculator use the `number` class, which frees it from the `number_impl` class. Implement derived classes for the types you want to support: integer, double, rational, etc.

As you can see, this kind of project can continue as long as you want it to. There will always be another feature to add. Just be sure to add features in small increments.

Similarly, your journey toward C++ expertise never ends. There will always be new surprises—waiting just around the corner, in the middle of your next project, with the next compiler upgrade. As I write this, the standardization committee is finishing work on the next version of the C++ language standard, C++ 14. After that will come the next language-revision cycle, and the next, and the next.

I wish you luck on your voyage, and I hope you enjoy the explorations to come.

Index

Exploring C++ 11

Problems and Solutions Handbook

Ray Lischner

Apress®

Exploring C++ 11: Problems and Solutions Handbook

Copyright © 2013 by Ray Lischner

ISBN-13 (pbk): 978-1-4302-6193-3

ISBN-13 (electronic): 978-1-4302-6194-0

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science+Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Contents

About the Author

All the world is paged,

And all the men and women merely programs:

They have their exits and their segfaults;

And one man in his time plays many games,

His acts being seven ages. At first, the newbie,

Mewling and puking in BASIC terms.

And then the whining school-boy, with his packages,

And JavaServer Faces, creeping like snail

Downloading from the Web. And then the l0v3r,

Sighing like heat sink fan, with an unmerged commit

Made to his github project. Then a hacker,

Full of strange oaths and bearded like a guru,

Jealous in honor, sudden and quick in quarrel,

Seeking the flamebait reputation

Even on lkml. And then the justice,

In fair round belly with cappuccino drowned,

With eyes severe and beard of two days' cut,

Full of wise saws and modern design patterns;

And so he plays his part. The sixth age shifts

Into the lean and sandal'd pantaloon,

With bifocals on nose and balding pate,

His COBOL code, well saved, a world too wide

For his shrunk shank; and his big noisy voice,

Turning again toward childish errors, buffer

Overruns in his code. Last scene of all,

That ends this strange eventful history,

Is second childishness and mere oblivion, Sans mouse, sans keyboard, sans debugger, sans everything.

By William Shakespeare, edited by Ray Lischner

Ray Lischner started writing programs before he had access to a computer, and over the subsequent three decades, he progressed steadily through the ages of programming. He currently lives with his wife, children, and four-terabyte MythTV server in Maryland, where he does his best to retard the inexorable descent into the seventh age.

About the Technical Reviewers

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for Brain Force (`www.brainforce.com`) in its Italian branch (`www.brainforce.it`). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past ten years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics.

Stefan Turalski is just another coder, who is trying to optimize his small world full of libraries, servers, tests, config files, and the like a line of code at a time.

Wearing many hats, with well over a decade of experience in software development, he specializes in implementation, design, and team management.

He has built a wide range of systems, from embedded networking, health care, and low-latency trading to highly scalable, distributed C++ / Java / .NET enterprise-class solutions. Stefan is currently working at a financial institution in London.

Acknowledgments

As with any book, many hardworking folk contributed to the product you hold in your hands (or read online). I thank the technical reviewers, Stefan Turalski and Fabio Claudio Ferracchiati, for their many corrections, suggestions, and comments. I thank the Apress editors and staff, without whom this book would be no more than bits on my file server: Ewan Buckingham, Katie Sullivan, Christine Ricketts, and Michael G. Laraque. Most of all, I thank my wife, Cheryl, and my children for their patience with my repeated absence on evenings and weekends.

Finally, I thank my readers, for they are the most important people in the publishing business.