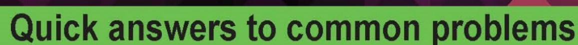


C++ Game Development Cookbook

Over 90 recipes to get you creating modern, fast, and high-quality games with C++

Druhin Mukherjee

[PACKT] PUBLISHING open source*
community experience distilled

www.allitebooks.com

C++ Game Development Cookbook

Over 90 recipes to get you creating modern, fast, and high-quality games with C++

Druhin Mukherjee

BIRMINGHAM - MUMBAI

www.allitebooks.com

C++ Game Development Cookbook

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2016

Production reference: 1250516

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-272-2

www.packtpub.com

www.allitebooks.com

Credits

Author
Druhin Mukherjee

Reviewer
Gonzalo Peces Nicolás

Acquisition Editor
Kirk D'costa

Content Development Editor
Rashmi Suvarna

Technical Editor
Anushree Arun Tendulkar

Copy Editor
Safis Editing

Project Coordinator
Judie Jose

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

www.allitebooks.com

About the Author

Druhin Mukherjee is currently the co-founder and technical Director at GodSpeed Games. He has, over the years, worked with clients such as Lightning Fish Games, Chromativity, Rockstar North, Tag Games, BBC, Dynamo Games, and Codemasters.

Druhin has been balancing making games and teaching video game programming to enthusiastic students. He spent 3 years in Auckland, New Zealand as a Senior Lecturer in the Games department at Media Design School.

As a passionate games developer, Druhin has been sharing his knowledge on the Internet as blogs and websites. His recently started website for solving game development puzzles has over thousand subscribers.

Druhin has collaborated with other writers and published many journals and papers; however, this is his first official effort to write a book.

I would like to thank my wife, Anushree, for putting up with my late night writing sessions. I also give deep thanks and gratitude to Rashmi Suvarna, without whose efforts this book quite possibly would not have happened.

I would also like to thank all of the mentors that I've had over the years. Without learning from these teachers, there is not a chance I could be doing what I do today, and it is because of them and others that I feel compelled to pass my knowledge on to those willing to learn.

www.allitebooks.com

About the Reviewer

Gonzalo Peces Nicolás is a Senior Game Developer based in Hong Kong. Gonzalo received his Bachelor's Degree in Computer Science in 2005 in Spain and his Master's Degree in Computer Games Development in 2011 in Scotland. Currently, he is working as Senior Software Engineer in one of the major Game Mobile publishers in Asia. He has, over the years, been involved in numerous games in some of the most prolific international game companies in Europe and Asia, developing on multiple platforms, such as PC, Mac, mobile, and console.

Furthermore, he has over a decade of professional development in several industries, which includes not only game development but also telecommunications and cryptography.

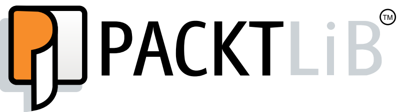
www.allitebooks.com

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

www.allitebooks.com

Table of Contents

Preface v

Chapter 1: Game Development Basics 1

- Introduction 1
- Installing an IDE on Windows 2
- Choosing the right source control tool 5
- Using call stacks for memory storage 7
- Using recursions cautiously 9
- Using pointers to store memory addresses 11
- Casting between different datatypes 14
- Managing memory more effectively using dynamic allocation 16
- Using bitwise operations for advanced checks and optimization 21

Chapter 2: Object-Oriented Approach and Design in Games 25

- Introduction 25
- Using classes for data encapsulation and abstraction 26
- Using polymorphism to reuse code 30
- Using copy constructors 34
- Use operator overloading to reuse operators 36
- Use function overloading to reuse functions 45
- Using files for input and output 48
- Creating your first simple game 52
- Templates – when to use them 55

Chapter 3: Data Structures in Game Development 59

- Introduction 59
- Using more advanced data structures 60
- Using linked lists to store data 70
- Using stacks to store data 72
- Using queues to store data 75
- Using trees to store data 77

www.allitebooks.com

Using graphs to store data 81
Using STL lists to store data 83
Using STL maps to store data 85
Using STL hash tables to store data 86

Chapter 4: Algorithms for Game Development 89
Introduction 89
Using sorting techniques to arrange items 90
Using searching techniques to look for an item 93
Finding the complexity of an algorithm 95
Finding the endian-ness of a device 97
Using dynamic programming to break down a complex problem 99
Using greedy algorithms to solve problems 101
Using divide and conquer algorithms to solve problems 102

Chapter 5: Event-Driven Programming – Making Your First 2D Game 107
Introduction 107
Starting to make a Windows game 108
Using Windows classes and handles 109
Creating your first window 114
Adding keyboard and mouse controls with text output 118
Using Windows resources with GDI 126
Using dialogs and controls 131
Using sprites 136
Using animated sprites 155

Chapter 6: Design Patterns for Game Development 159
Introduction 159
Using the singleton design pattern 160
Using the factory method 162
Using the abstract factory method 165
Using the observer pattern 170
Using the flyweight pattern 174
Using the strategy pattern 179
Using the command design pattern 184
Creating an advanced game using design patterns 187

Chapter 7: Organizing and Backing Up 191
Introduction 191
Versions of source control 192
Installing a versioning client 192
Selecting a host to save your data 193
Adding source control – committing and updating your code 195

www.allitebooks.com

Resolving conflicts 197
Creating a branch 198
Chapter 8: AI in Game Development 201
Introduction 201
Adding artificial intelligence to a game 202
Using heuristics in a game 203
Using a Binary Space Partition Tree 205
Creating a decision making AI 208
Adding behavioral movements 219
Using neural network 222
Using genetic algorithms 227
Using other waypoint systems 231
Chapter 9: Physics in Game Development 233
Introduction 233
Using physics rules in your game 234
Making things collide 239
Installing and integrating Box2D 245
Making a basic 2D game 246
Making a 3D game 250
Creating a particle system 252
Using ragdoll in your game 254
Chapter 10: Multithreading in Game Development 263
Introduction 263
Concurrency in games – creating a thread 264
Joining and detaching a thread 265
Passing arguments to a thread 266
Avoiding deadlocks 268
Data race and mutex 269
Writing a thread-safe class 271
Chapter 11: Networking in Game Development 275
Introduction 276
Understanding the different layers 276
Selecting the appropriate protocol 278
Serializing the packets 281
Using socket programming in games 286
Sending the data 289
Receiving the data 294
Dealing with lag 297
Using synchronized simulation 299

www.allitebooks.com

Using area of interest filtering 301
Using local perception filter 302
Chapter 12: Audio in Game Development 305
Introduction 305
Installing FMOD 306
Adding background music 306
Adding sound effects 308
Creating a sound effect manager 309
Dealing with multiple sound file names 312
Chapter 13: Tips and Tricks 315
Introduction 315
Effectively commenting your code 315
Using bit fields in a struct 317
Writing a sound technical design document 319
Using the const keyword to optimize your code 320
Using bit shift operators in an enum 322
Using the new lambda function of C++ 11 323
Index 325

Preface

This book provides a detailed look at some of the aspects of C++ which could be used for games development.

What this book covers

Chapter 1, *Game Development Basics*, explains the basics of C++ programming, writing small programs to be used in games, and how to handle memory in games.

Chapter 2, *Object-Oriented Approach and Design in Games*, explains the use OOP concepts in games, and you will make a small prototype text-based game.

Chapter 3, *Data Structures in Game Development*, introduces all the simple and complex data structures in C++ and shows how to use them effectively in games.

Chapter 4, *Algorithms for Game Development*, explains various algorithms that can be used in games. It also covers means to measure the efficiency of an algorithm.

Chapter 5, *Event-Driven Programming – Making Your First 2D Game*, introduces Windows programming, creating sprites, and animation.

Chapter 6, *Design Patterns for Game Development*, explains how to use well-known design patterns in game development and when not to use them.

Chapter 7, *Organizing and Backing Up*, explains the importance of backing up data and the importance of sharing data across a team.

Chapter 8, *AI in Game Development*, explains how to approach writing artificial intelligence in games.

Chapter 9, *Physics in Game Development*, explains how to make bodies collide and how to use third-party physics libraries, such as Box2D, to make games.

Chapter 10, *Multithreading in Game Development*, explains how to use the thread architecture of C++11 to make games.

Chapter 11, *Networking in Game Development*, explains the fundamentals of writing a multiplayer game.

Chapter 12, *Audio in Game Development*, explains how to add sound and music effects to games, and avoiding memory leaks while playing sounds.

Chapter 13, *Tips and Tricks*, has some neat tips and tricks of using C++ to make games.

What you need for this book

For this book you would require a Windows machine and a working copy of Visual Studio 2015 Community Edition.

Who this book is for

This book should be primarily used by college students wanting to enter the games industry or enthusiastic school students who want to get their hands dirty early and understand the fundamentals of game programming. This book also has some very technical chapters which will be very useful for industry professionals for reference or to keep by the side while solving complex problems.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

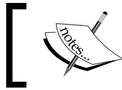
In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "If you have a file called `main.cpp`, it will generate an object code called `main.o`."

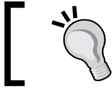
A block of code is set as follows:

```
#include <iostream>
#include <conio.h>

using namespace std;

int countTotalBullets(int iGun1Ammo, int iGun2Ammo)
{
    return iGun1Ammo + iGun2Ammo;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on **Download Visual Studio Community**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.

5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/C++Game-Development-Cookbook`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Game Development Basics

In this chapter, the following recipes will be covered:

- Installing an IDE on Windows
- Choosing the right source control tool
- Using call stacks for memory storage
- Using recursions cautiously
- Using pointers to store memory addresses
- Casting between various datatypes
- Managing memory more effectively using dynamic allocation
- Using bitwise operations for advanced checks and optimization

Introduction

In this chapter, we will cover the basic concepts that you need to know to kick-start your career in game development.

The first step before a person starts coding is to install an **integrated development environment** (**IDE**). Nowadays, there are a few online IDEs that are available, but we are going to use an offline standalone IDE, **Visual Studio**. The next most important thing that many programmers do not start using at an early stage is **revision control software**.

Revision control software helps to back up the code in one central location; it has a historical overview of the changes that are made, which you can access and revert to if needed, and it also helps to resolve conflicts between files that have been worked on by different programmers at the same time.

The most useful feature of C++, in my opinion, is **memory handling**. It gives the developers a lot of control over how memory must be assigned depending on the current usage and needs of the program. As a result of this, we can allocate memory when there is a need and deallocate it accordingly.

If we do not de-allocate memory, we might run out of memory very soon, especially if we are using recursion. Sometimes there is a need to convert from one datatype to another to prevent loss of data, to pass the correct datatype in a function, and so on. C++ provides us a few ways by which we can do those castings.

The recipes in this chapter will primarily focus on these topics and deal with practical ways to implement them.

Installing an IDE on Windows

In this recipe, we will find out how easy it is to install Visual Studio on your Windows machine.

Getting ready

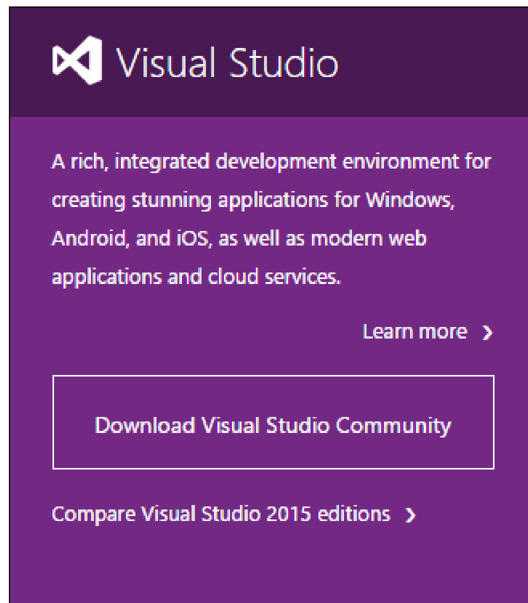
To go through this recipe, you will need a machine running Windows. No other prerequisites are required.

How to do it...

Visual Studio is a powerful IDE in which most professional software is written. It has loads of features and plugins to help us write better code:

1. Go to `https://www.visualstudio.com`.

2. Click on **Download Visual Studio Community**.



Download Visual Studio Community

3. This should download an `.exe` file.
4. After it downloads, double-click on the setup file to start the installation.
5. Make sure you have all the updates necessary on your Windows machine.
6. You can also download any version of Visual Studio or Visual C++ Express.
7. If the application asks for starting environment settings, select **C++** from the available options.

A few things to note are listed here:

- You need a Microsoft account to install it.
- There are other free IDEs for C++, such as **NetBeans**, **Eclipse**, and **Code::Blocks**.
- While Visual Studio works only for Windows, Code::Blocks and other such IDEs are cross-platform and can work on Mac and Linux as well.

For the remainder of this chapter, all code examples and snippets will be provided using Visual Studio.

How it works...

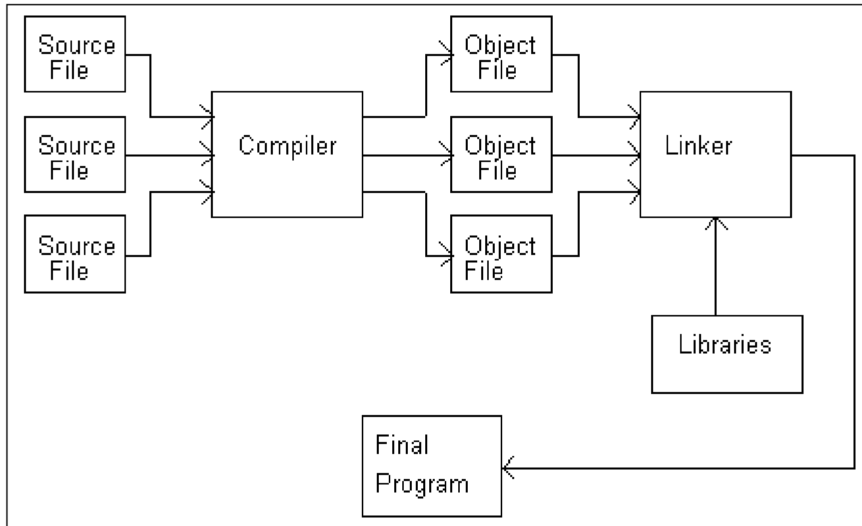
An IDE is a programming environment. An IDE consists of various functionalities that can vary from one IDE to another. However, the most basic functionalities that are present in all IDEs are a code editor, a compiler, a debugger, a linker, and a GUI builder.

A code editor, or a source code editor as they are otherwise known, is useful for editing code written by programmers. They provide features such as auto-correct, syntax highlighting, bracket completion and indentation, and so on. An example snapshot of the Visual Studio code editor is shown here:

```
91        case WM_COMMAND:
92           {
93              WORD hi = HIWORD(wParam);
94              WORD lo = LOWORD(wParam);
95
96              if(hi == 0 || hi == 1)
97                 OnMenuItemClicked(lo);
98           }
99           break;
100       }
101
102       return ::DefWindowProc(hWnd, message, wParam, lParam);
103    }
104
105    Window* WindowFromHandler(HWND const hWnd)
106    {
107       auto result = g_wndMap.find(hWnd);
108       return result == g_wndMap.end() ? nullptr : result->second;
109    }
110
111    LRESULT CALLBACK CallWinProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
112    {
113       auto wnd = WindowFromHandler(hWnd);
114       if(wnd == nullptr || (HWND)wnd == nullptr)
115          return ::DefWindowProc(hWnd, message, wParam, lParam);
116
117       return wnd->WndProc(message, wParam, lParam);
118    }
```

A **compiler** is a computer program that converts your C++ code to object code. This is necessary in order to create an executable. If you have a file called `main.cpp`, it will generate an object code called `main.o`.

A **linker** is a computer program that converts the object code generated by the compiler to an executable or a library file:



Compiler and linker

A **debugger** is a computer program that helps to test and debug computer programs.

A **GUI builder** helps the designer and programmer to create GUI content or widgets easily. It uses a drag and drop **WYSIWYG** tool editor.

Choosing the right source control tool

In this recipe, we will see how easy it is to take a backup of our code using the right version control. The advantages of having a backup to a central server is that you will never lose work, can download the code on any machine, and can also go back to any of your changes from the past. Imagine it is like a checkpoint that we have in games, and you can go back to that checkpoint if you face problems.

Getting ready

To go through this recipe, you will need a machine running Windows. No other prerequisites are required.

How to do it...

Choosing a correct version tool is very important as it will save a lot of time organizing data. There are a few versioning tools that are available, so it is very important that we should be informed about all of them so that we can use the correct one based on our needs.

First analyze the choices that are available to you. The choices primarily include **Concurrent Versions System** (**CVS**), **Apache Subversion** (**SVN**), **Mercurial**, and **GIT**.

How it works...

CVS has been around for a long time, so there is tons of documentation and help available. However, a lack of atomic operations often leads to source corruption and it is not well cut out for long-term branching operations.

SVN was made as an improvement to CVS and it does fix many of its issues relating to atomic operations and source corruption. It is free and open source. It has lots of plugins for different IDEs. However, one of the major drawbacks of this tool is that it is comparatively very slow in its operations.

GIT was made primarily for Linux but it improves operation speed a lot. It works on UNIX systems as well. It has cheap branch operations but it is not totally optimized for a single developer and its Windows support is limited compared to Linux. However, GIT is extremely popular and many prefer GIT to SVN or CVS.

Mercurial came into existence shortly after GIT. It has node-based operations but does not allow the merging of two parent branches.

So to sum up, use SVN if you want a central repository that others can push and pull. Although it has its limitations, it's easy to learn. Use Mercurial or GIT if you want a distributed model. In this case, there is a repository on every computer, and generally, one of them is regarded as the *official* one. Mercurial is often preferred if it is a relatively small team, and it's easier to learn than GIT.

We will look into these in more detail in a separate chapter.

Detailed steps to download the code bundle are mentioned in the Preface of this book. Please have a look.

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/C++Game-Development-Cookbook`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Using call stacks for memory storage

The main reason why C++ is still the preferred language for most game developers is that you handle memory yourself and control the allocation and de-allocation of memory to a great extent. For that reason, we need to understand the different memory spaces that are provided to us. When data is "pushed" onto the stack, the stack grows. As data is "popped" off the stack, the stack shrinks. It is not possible to pop a particular piece of data off the stack without first popping off all data placed on top of it. Think of this as a series of compartments aligned top to bottom. The top of the stack is whatever compartment the stack pointer happens to point to (this is a register).

Each compartment has a sequential address. One of those addresses is kept in the stack pointer. Everything below that magic address, known as the top of the stack, is considered to be on the stack. Everything above the top of the stack is considered to be off the stack. When data is pushed onto the stack, it is placed into a compartment above the stack pointer, and then the stack pointer is moved to the new data. When data is popped off the stack, the address of the stack pointer is changed by moving it down the stack.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

C++ is probably one of the best programming languages out there and one of the main reasons for that is that it is also a low level language, because we can manipulate memory. To understand memory handling, it is very important to understand how memory stacks work:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `main.cpp` or anything that you want to name the source file.
5. Add the following lines of code:

```
#include <iostream>
#include <conio.h>

using namespace std;

int countTotalBullets(int iGun1Ammo, int iGun2Ammo)
{
```

```
        return iGun1Ammo + iGun2Ammo;
    }

    int main()
    {
        int iGun1Ammo = 3;
        int iGun2Ammo = 2;
        int iTotalAmmo = CountTotalBullets(iGun1Ammo,
            iGun2Ammo);

        cout << "Total ammunition currently with you
            is"<<iTotalAmmo;

        _getch();
    }
```

How it works...

When you call the function `CountTotalBullets`, the code branches to the called function. The parameters are passed in and the body of the function is executed. When the function completes, a value is returned and the control returns to the calling function.

But how does it really work from a compiler's point of view? When you begin your program, the compiler creates a stack. The **stack** is a special area of memory allocated for your program in order to hold the data for each function in your program. A stack is a **Last In First Out** (**LIFO**) data structure. Imagine a deck of cards; the last card put on the pile will be the first card taken out.

When your program calls `CountTotalBullets`, a stack frame is established. A **stack frame** is an area of the stack set aside to manage that function. This is very complex and different on different platforms, but these are the essential steps:

1. The return address of `CountTotalBullets` is put on the stack. When the function returns, it will resume executing at this address.
2. Room is made on the stack for the return type you have declared.
3. All arguments to the function are placed on the stack.
4. The program branches to your function.
5. Local variables are pushed onto the stack as they are defined.

Using recursions cautiously

Recursions are a form of programming design in which the function calls itself multiple times to solve a problem by breaking down a large solutions set into multiple small solution sets. The code size definitely shortens. However, if not used properly, recursions can fill up the call stack really fast and you can run out of memory.

Getting ready

To get started with this recipe, you should have some prior knowledge of call stacks and how memory is assigned during a function call. You need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, you will see how easy it is to use recursions. Recursions are very smart to code but also can lead to some serious problems:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `main.cpp` or anything that you want to name the source file.
5. Add the following lines of code:

```
#include <iostream>
#include <conio.h>

using namespace std;
int RecursiveFactorial(int number);
int Factorial(int number);
int main()
{
    long iNumber;
    cout << "Enter the number whose factorial you want
      to find";
    cin >> iNumber;

    cout << RecursiveFactorial(iNumber) << endl;
```

```
    cout << Factorial(iNumber);

    _getch();
    return 0;
}

int Factorial(int number)
{
    int iCounter = 1;
    if (number < 2)
    {
        return 1;
    }
    else
    {
        while (number>0)
        {
            iCounter = iCounter*number;
            number -= 1;
        }

    }
    return iCounter;
}

int RecursiveFactorial(int number)
{
    if (number < 2)
    {
        return 1;
    }
    else
    {
        while (number>0)
    {
            return number*Factorial(number - 1);
        }
    }

}
```

How it works...

As you can see from the preceding code, both the functions find the factorial of a number. However, when using recursion, the stack size will grow immensely with each function call; the stack pointer has to be updated every call and data pushed onto the stack. With recursion, as the function calls itself, every time a function is called from within itself the stack size will keep on rising until it runs out of memory and creates a deadlock or crashes.

Imagine finding the factorial of 1000. The function will be called within itself a very large number of times. This is a recipe for certain disaster and we should avoid such coding practices to a great extent.

There's more...

You can use a larger datatype than int if you are finding the factorial of a number greater than 15, as the resulting factorial will be too large to be stored in int.

Using pointers to store memory addresses

In the previous two recipes, we have seen how not having sufficient memory can be a problem to us. However, until now, we have had no control over how much memory is assigned and what is assigned to each memory address. Using pointers, we can address this issue. In my opinion, pointers are the single most important topic in C++. If your concept of C++ has to be clear, and if you are to become a good developer in C++, you must be good with pointers. Pointers can seem very daunting at first, but once you get the hang of it, pointers are pretty easy to use.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will see how easy it is to work with pointers. Once you are comfortable using pointers, we can manipulate memory and store references in memory quite easily:

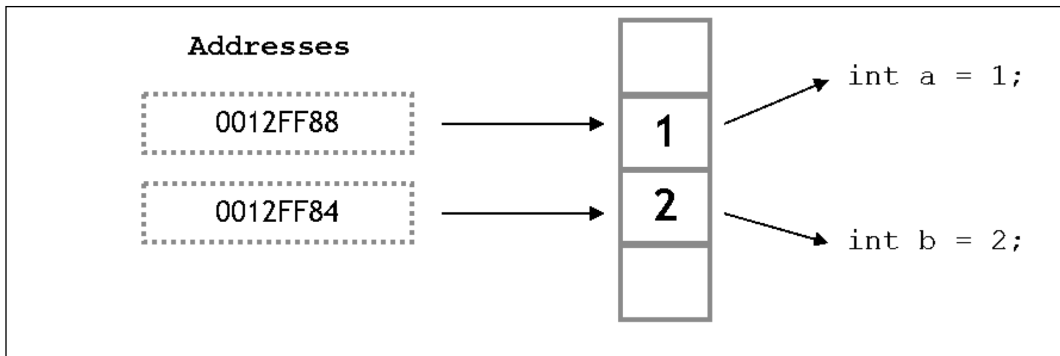
1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `main.cpp` or anything that you want to name the source file.

5. Add the following lines of code:

```
#include <iostream>
#include <conio.h>

using namespace std;

int main()
{
    float fCurrentHealth = 10.0f;

    cout << "Address where the float value is stored: "
        << &fCurrentHealth << endl;
    cout << "Value at that address: "
        << *(&fCurrentHealth) << endl;

    float* pfLocalCurrentHealth = &fCurrentHealth;
    cout << "Value at Local pointer variable:
        "<<pfLocalCurrentHealth << endl;
    cout << "Address of the Local pointer variable:
        "<<&pfLocalCurrentHealth << endl;
    cout << "Value at the address of the Local pointer
        variable: "<<*pfLocalCurrentHealth << endl;

    _getch();
    return 0;
}
```

How it works...

One of the most powerful tools of a C++ programmer is to manipulate computer memory directly. A **pointer** is a variable that holds a memory address. Each variable and object used in a C++ program is stored in a specific place in memory. Each memory location has a unique address. Memory addresses will vary depending on the operating system used. The amount of bytes taken up depends on the variable type: *float = 4 bytes*, *short = 2 bytes*:



Pointers and memory storage

Each location in the memory is 1 byte. The pointer `pfLocalCurrentHealth` holds the address of the memory location that has stored `fCurrentHealth`. Hence, when we display the contents of the pointer, we get the same address as that of the address containing the `fCurrentHealth` variable. We use the `&` operator to get the address of the `pfLocalCurrentHealth` variable. When we reference the pointer using the `*` operator, we get the value stored at the address. Since the stored address is same as the address storing `fCurrentHealth`, we get the value `10`.

There's more...

Let us consider the following declarations:

- `const float* pfNumber1`
- `float* const pfNumber2`
- `const float* const pfNumber3`

All of these declarations are valid. But what do they mean? The first declaration states that `pfNumber1` is a pointer to a constant float. The second declaration states that `pfNumber2` is a constant pointer to a float. The third declaration states that `pfNumber3` is a constant pointer to a constant integer. The key differences between references and these three types of const pointers are listed here:

- `const` pointers can be NULL
- A reference does not have its own address, whereas a pointer does

  The address of a reference is the actual object's address
- A pointer has its own address and it holds as its value the address of the value it points to

www.allitebooks.com

For more information on pointers and references, go to the following link:
http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in/57492#57492

Casting between different datatypes

Casting is a conversion process of changing some data into a different type of data. We can convert between built-in types or our own datatypes. Some of the conversions are done automatically by the compiler, and the programmer does not have to intervene. Such conversions are called **implicit conversions**. Other conversions, which have to be directly specified by the programmer, are called explicit conversion. Sometimes we may get warnings about *loss of data*. We should pay heed to these warnings and think about how this might adversely affect our code. Casting is commonly used when the interface expects a particular type, but we want to feed it data of a different type. With C, we can cast anything to everything. However, C++ provides us with finer controls.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will see how we can easily cast or convert between various datatypes. Usually, a programmer uses C-style casting even in C++, but this is not recommended. C++ provides us with its own style of casting for different situations which we should use:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `main.cpp` or anything that you want to name the source file.

5. Add the following lines of code:

```
#include <iostream>
#include <conio.h>

using namespace std;

int main()
{
    int iNumber = 5;
    int iOurNumber;
    float fNumber;

    //No casting. C++ implicitly converts the result
      into an int and saves
    //into a float
    fNumber = iNumber/2;
    cout << "Number is " << fNumber<<endl;

    //C-style casting. Not recommended as this is not type safe
    fNumber = (float)iNumber / 2;
    cout << "Number is " << fNumber<<endl;

    //C++ style casting. This has valid constructors to make the
casting a safe one
    iOurNumber = static_cast<int>(fNumber);
    cout << "Number is " << iOurNumber << endl;

    _getch();
    return 0;
}
```

How it works...

There are four types of casting operators in C++, depending on what we are casting: `static_cast`, `const_cast`, `reinterpret_cast`, and `dynamic_cast`. Now, we are going to look at `static_cast`. We will look at the remaining three casting technique after we discuss dynamic memory and classes. Converting from a smaller datatype to a larger type is called promotion and is guaranteed to have no data loss. However, conversion from a larger datatype to a smaller one is called demotion and may lead to data loss. Compilers will generally give you a warning when this happens, and you should pay heed to this.

Let us look at the previous example. We have initialized an integer with the value `5`. Next, we have initialized a floating point variable and stored the result of `5` divided by `2`, which is `2.5`. However, when we display the variable `fNumber`, we see that the displayed value is `2`. The reason is the C++ compiler implicitly casts the result of `5/2` and stores it as an integer. So it is evaluating something similar to int (`5/2`) which is int (`2.5`), evaluating to `2`. So to achieve our desired result, we have two options. The first method is a C-style explicit cast, which is not recommended at all because it does not have a type safe check. The format for the C-style cast is (`resultant_data_type`) (`expression`), which in this case is something like float (`5/2`). We are explicitly telling the compiler to store the result of the expression as a floating point number. The second method, and a more C++ style way of doing the cast, is by using the `static_cast` operation. This has suitable constructors to dictate that the conversion is type safe. The format for a `static_cast` operation is `static_cast<resultant_data_type>(expression)`. The compiler checks if the casting conversion is safe and then executes the type casting operation.

Managing memory more effectively using dynamic allocation

Programmers generally deal with five areas of memory: **global namespace**, **registers**, **code space**, **stack**, and the **free store**. When an array is initialized, the number of elements has to be defined. This leads to lots of memory problems. Most of the time, not all elements that we allocated are used, and sometimes we need more elements. To help overcome this problem, C++ facilitates memory allocation while an `.exe` file is running by using the free store.

The free store is a large area of memory that can be used to store data, and is sometimes referred to as *the heap*. We can request some space on the free store, and it will give us an address that we can use to store data. We need to keep that address in a pointer. The free store is not cleaned up until your program ends. It is the programmer's responsibility to free any free store memory used by their program.

The advantage of the free store is that there is no need to preallocate all variables. We can decide at runtime when more memory is needed. The memory is reserved and remains available until it is explicitly freed. If memory is reserved while in a function, it is still available when control returns from that function. This is a much better way of coding than global variables. Only functions that have access to the pointer can access the data stored in memory, and it provides a tightly controlled interface to that data.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

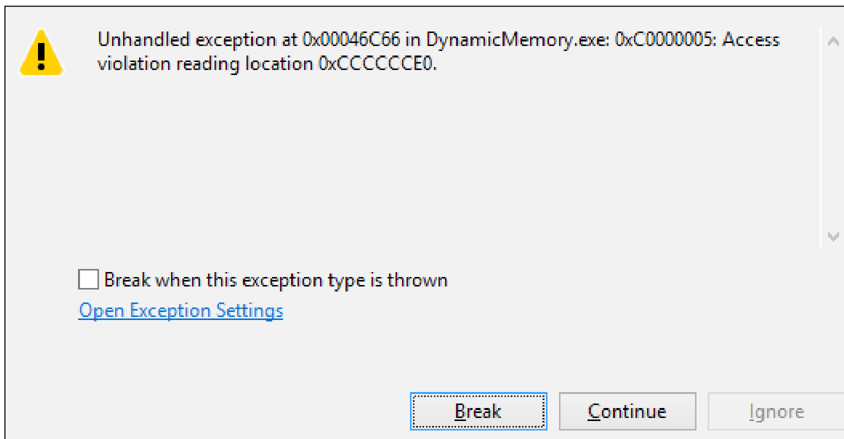
In this recipe, we will see how easy it is to use dynamic allocation. In games, most of the memory is allocated dynamically at runtime as we are never sure how much memory we should assign. Assigning an arbitrary amount of memory may result in less memory or memory wastage:

1. Open Visual Studio.
2. Create a new C++ project.
3. Add a source file called `main.cpp` or anything that you want to name the source file.
4. Add the following lines of code:

```
#include <iostream>
#include <conio.h>
#include <string>

using namespace std;

int main()
{

  int iNumberofGuns, iCounter;
  string * sNameOfGuns;
  cout << "How many guns would you like to purchase? ";
  cin >> iNumberofGuns;
  sNameOfGuns = new string[iNumberofGuns];
  if (sNameOfGuns == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (iCounter = 0; iCounter<iNumberofGuns; iCounter++)
    {
      cout << "Enter name of the gun: ";
      cin >> sNameOfGuns[iCounter];
    }
    cout << "You have purchased: ";
    for (iCounter = 0; iCounter<iNumberofGuns; iCounter++)
      cout << sNameOfGuns[iCounter] << ", ";
    delete[] sNameOfGuns;
  }

  _getch();
  return 0;
}
```

How it works...

You can allocate memory to the free store using the `new` keyword; `new` is followed by the type of the variable you want to allocate. This allows the compiler to know how much memory will need to be allocated. In our example, we have used string. The `new` keyword returns a memory address. This memory address is assigned to a pointer, `sNameOfGuns`. We must assign the address to a pointer, otherwise the address will be lost. The format for using the `new` operator is `datatype * pointer = new datatype`. So in our example, we have used `sNameOfGuns = new string[iNumberofGuns]`. If the new allocation fails, it will return a null pointer. We should always check whether the pointer allocation has been successful; otherwise we will try to access a part of the memory that has not been allocated and we may get an error from the compiler, as shown in the following screenshot, and your application will crash:



When you are finished with the memory, you must call delete on the pointer. Delete returns the memory to the free store. Remember that the pointer is a local variable. Where the function that the pointer is declared in goes out of scope, the memory on the free store is not automatically deallocated. The main difference between static and dynamic memory is that the creation/deletion of static memory is handled automatically, whereas dynamic memory must be created and destroyed by the programmer.

The `delete[]` operator signals to the compiler that it needs to free an array. If you leave the brackets off, only the first element in the array will be deleted. This will create a memory leak. Memory leaks are really bad as it means there are memory spaces that have not been deallocated. Remember, memory is a finite space, so eventually you are going to run into trouble.

When we use `delete[]`, how does the compiler know that it has to free *n* number of strings from the memory? The runtime system stores the number of items somewhere it can be retrieved only if you know the pointer `sNameOfGuns`. There are two popular techniques that do this. Both of these are used by commercial compilers, both have tradeoffs, and neither are perfect:

- Technique 1:

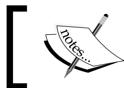
  Over-allocate the array and put the number of items just to the left of the first element. This is the faster of the two techniques, but is more sensitive to the problem of the programmer saying `delete sNameOfGuns`, instead of `delete[] sNameOfGuns`.

- Technique 2:

  Use an associative array with the pointer as a key and the number of items as the value. This is the slower of the two techniques, but is less sensitive to the problem of the programmer saying `delete sNameOfGuns`, instead of `delete[] sNameOfGuns`.

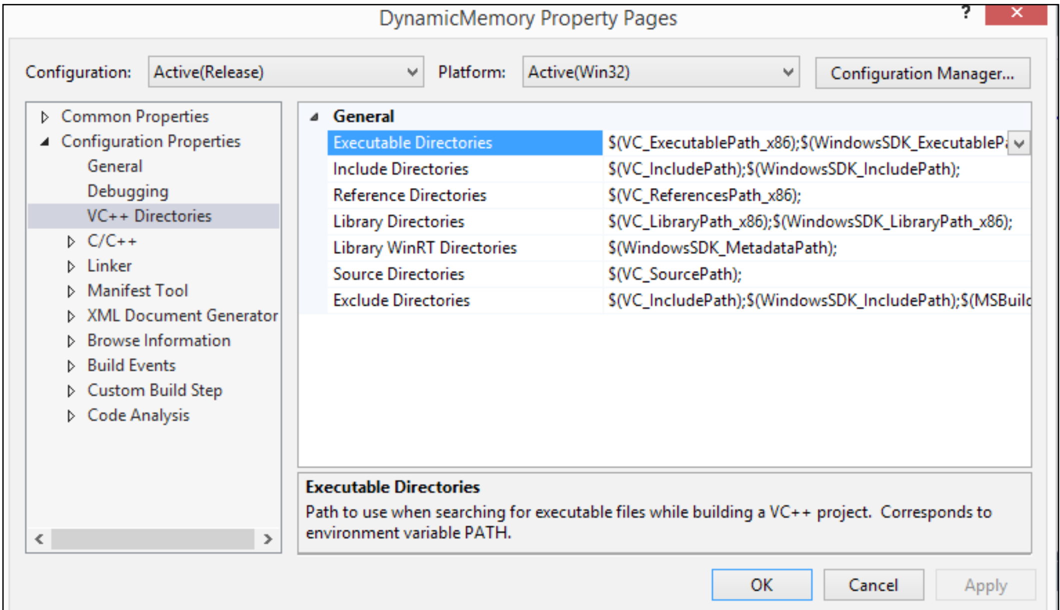
There's more...

We can also use a tool called **VLD** to check for memory leaks.



Download VLD from `https://vld.codeplex.com/`.

After the setup has downloaded, install VLD on your system. This may or may not set up the VC++ directories correctly. If it doesn't, do it manually by right-clicking on the project page and adding the directory of VLD to the field called **Include Directories**, as shown in the following figure:



After setting up the directories, add the header file `<vld.h>` in your source file. After you execute your application and exit it, your output window will now show whether there are any memory leaks in your application.

Understanding the error messages

When using the debug build, you may notice the following values in memory during debugging:

- `0xCCCCCCCC`: This refers to values being allocated on the stack, but not yet initialized.
- `0xCDCDCDCD`: This means memory has been allocated in the heap, but it is not yet initialized (clean memory).
- `0xDDDDDDDD`: This means memory has been released from the heap (dead memory).
- `0xFEEEFEEE`: This refers to values being deallocated from the free store.
- `0xFDFDFDFD`: "No man's land" fences, which are placed at the boundary of heap memory in debug mode. These should never be overwritten, and if they are, it probably means the program is trying to access memory at an index outside of an array's max size.

Using bitwise operations for advanced checks and optimization

In most cases, a programmer will not need to worry too much about bits unless there is a need to write some compression algorithms, and when we are making a game, we never know when a situation such as that arises. In order to encode and decode files compressed in this manner, you need to actually extract data at the bit level. Finally, you can use bit operations to speed up your program or perform neat tricks. However, this is not always recommended.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will see how easy it is to use bitwise operations to perform operations by manipulating memory. Bitwise operations are also a great way to optimize code by directly interacting with memory:

1. Open Visual Studio.
2. Create a new C++ project.
3. Add a source file called `main.cpp` or anything that you want to name the source file.
4. Add the following lines of code:

```
#include <iostream>
#include <conio.h>

using namespace std;

void Multi_By_Power_2(int iNumber, int iPower);
void BitwiseAnd(int iNumber, int iNumber2);
void BitwiseOr(int iNumber, int iNumber2);
void Complement(int iNumber4);
void BitwiseXOR(int iNumber,int iNumber2);

int main()
{
  int iNumber = 4, iNumber2 = 3;
  int iPower = 2;
  unsigned int iNumber4 = 8;

  Multi_By_Power_2(iNumber, iPower);
```

```
    BitwiseAnd(iNumber,iNumber2);
    BitwiseOr(iNumber, iNumber2);
    BitwiseXOR(iNumber,iNumber2);
    Complement(iNumber4);

    _getch();
    return 0;
}

void Multi_By_Power_2(int iNumber, int iPower)
{
    cout << "Result is :" << (iNumber << iPower)<<endl;
}
void BitwiseAnd(int iNumber, int iNumber2)
{
    cout << "Result is :" << (iNumber & iNumber2) << endl;
}
void BitwiseOr(int iNumber, int iNumber2)
{
    cout << "Result is :" << (iNumber | iNumber2) << endl;
}
void Complement(int iNumber4)
{
    cout << "Result is :" << ~iNumber4 << endl;
}
void BitwiseXOR(int iNumber,int iNumber2)
{
    cout << "Result is :" << (iNumber^iNumber2) << endl;
}
```

How it works...

The left shift operator is the equivalent of moving all the bits of a number a specified number of places to the left. In our example, the numbers we are sending to the function `Multi_By_Power_2` is `4` and `3`. The binary representation of `4` is `100`, so if we shift the most significant bit, which is `1`, three places to the left, we get `10000`, which is the binary of `16`. Hence, left shift is equivalent to integer division by `2^shift_arg`, that is, `4*2^3`, which is again `16`. Similarly, the right shift operation is equivalent to integer division by `2^shift_arg`.

Now let us consider we want to pack data so that the data is compressed. Consider the following example:

```
int totalammo,type,rounds;
```

We are storing the total bullets in a gun; the type of gun, but it can only be a rifle or pistol; and the total bullets per round it can fire. Currently we are using three integer values to store the data. However, we can compress all the preceding data into one single integer and hence compress the data:

```
int packaged_data;
packaged_data = (totalammo << 8) | (type << 7) | rounds;
```

If we assume the following notations:

- TotalAmmon: `A`
- Type: `T`
- Rounds: `R`

The final representation in the data would be something like this:

```
AAAAAAATRRRRRRR
```

2

Object-Oriented Approach and Design in Games

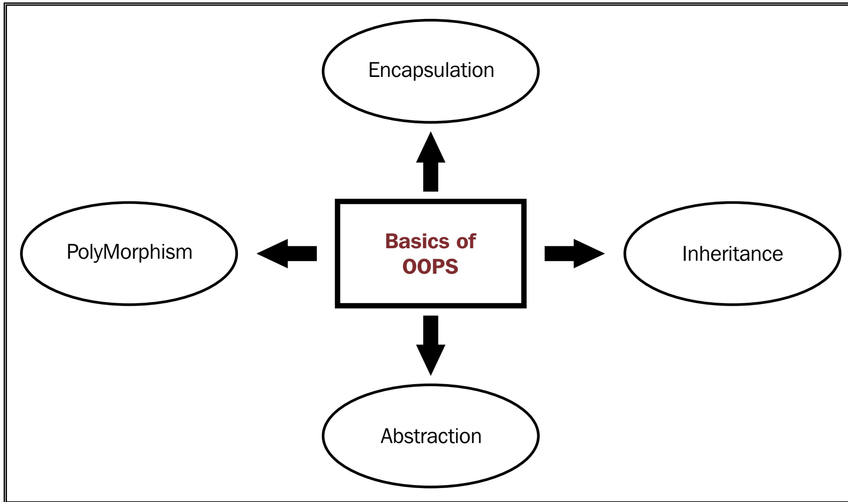
In this chapter, we will cover the following recipes:

- Using classes for data encapsulation and abstraction
- Using polymorphism to reuse code
- Using copy constructors
- Using operator overloading to reuse operators
- Using function overloading to reuse functions
- Using files for input and output
- Creating your first simple text-based game
- Templates – when to use them

Introduction

The following diagram shows the main concepts of **OOP** (**Object-oriented programming**). Let us consider that we need to make a car racing game. So, a car is made up of an engine, wheels, chassis, and so on. All these parts can be considered as individual components, which can be used for other cars as well. Similarly, every car's engine can be different and so we can add different functionalities, states, and properties to each individual component.

All this can be achieved through object-oriented programming:



We need to use an object-oriented system in any design that consists of states and behaviors. Let us consider a game like *Space Invaders*. The game consists of two main characters, the player ship and the enemy. There is also a boss, but that is just an advanced version of the enemy. The player ship can have different states such as alive, idle, moving, attack, and dead. It also has a few behaviors, such as left/right movement, single shoot/burst shoot/missile. Similarly, the enemy has states and behaviors. This is an ideal condition to use an object-oriented design. The boss is just an advanced form of the enemy, so we can use the concepts of polymorphism and inheritance to achieve the result.

Using classes for data encapsulation and abstraction

A class is used to organize information into meaningful states and behaviors. In games, we deal with so many different types of weapon, player, enemy, and terrain, each with its own type of state and behavior, so an object-oriented design with classes is a must.

Getting ready

To work through this recipe, you will need a machine running Windows. You need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to create a game framework using object-oriented programming in C++:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add source files called `Source.cpp`, `CEnemy.h`, and `CEnemy.cpp`.
5. Add the following lines of code to `Souce.cpp`:

```
#include "CEnemy.h"
#include <iostream>
#include <string>
#include <conio.h>
#include "vld.h"

using namespace std;

int main()
{
  CEnemy* pEnemy = new CEnemy(10,100,"DrEvil","GOLD");

  int iAge;
  int iHealth;
  string sName;
  string sArmour;

  iAge = pEnemy->GetAge();
  iHealth = pEnemy->TotalHealth();
  sArmour = pEnemy->GetArmourName();
  sName = pEnemy->GetName();

  cout << "Name of the enemy is :" << sName << endl;
  cout << "Name of " << sName << "'s armour is :" << sArmour <<
endl;
  cout << "Health of " << sName << " is :" << iHealth << endl;
  cout << sName << "'s age is :" << iAge;

delete pEnemy;
  _getch();
}
```

6. Add the following lines of code to `CEnemy.h`:

```
#ifndef _CENEMY_H
#define _CENEMY_H

#include <string>
using namespace std;

class CEnemy
{
public:
  string GetName()const;
  int GetAge()const;
  string GetArmourName()const;
  int TotalHealth()const;

  //ctors
  CEnemy(int,int,string,string);
//dtors
  ~CEnemy();
private:
  int m_iAge;
  int m_iHealth;
  string m_sName;
  string m_sArmour;
};

#endif
```

7. Add the following lines of code to `CEnemy.cpp`:

```
#include <iostream>
#include <string>
#include "CEnemy.h"

using namespace std;

CEnemy::CEnemy(int Age,int Health,int Armour,int Name)
{
  m_iAge = Age;
  m_iHealth = Health;
  m_sArmour = Armour;
  m_sName = Name;
}

int CEnemy::GetAge()const
```

```
{
  return m_iAge;
}

int CEnemy::TotalHealth()const
{
  return m_iHealth;
}

string CEnemy::GetArmourName()const
{
  return m_sArmour;
}

string CEnemy::GetName()const
{
  return m_sName;
}
```

How it works...

To create an object-oriented program, we need to create classes and objects. Although we can write the definition and declaration of a class in the same file, it is advisable to have two separate files for definition and declaration. A declaration class file is called a header file, whereas a definition class file is called a source file.

In the `CEnemy` header file, we define the member variables and the functions that we need. In a class, we have the option to separate out the variables as public, protected, or private. A public state indicates that they are accessible from outside the class, a protected state indicates that only the child class that inherits from the current base class has access to it, whereas a private state indicates that they are accessible by any instance of the class. By default, everything in a C++ class is private. Hence, we have created all the member functions as public so that we can access them from the driver program, which in this example is `Source.cpp`. The member variables in the header file are all private, as they should not be directly accessible from outside the class. This is what we call abstraction. We define a string type variable for name and armor, and an integer type for health and age. It is also advisable to create a constructor and destructor, even if we do not have any functionality for them at present. It is also good to use a copy constructor. The reason for this is explained later on in the chapter.

In the `CEnemy` source file, we have the initialization of the member variables and also the declarations of the functions. We have used the `const` keyword at the end of each function because we do not want the function to change the contents of the member variables. We just want them to return the values that are already assigned. As a rule of thumb, we should always use it unless it's necessary not to use it. It makes the code more secure, organized, and readable. We have initialized the variables in the constructor; we could have also created parameterized constructors and assigned values to them from the driver program. Alternatively, we can also have set functions to assign values.

From the driver program, we create a pointer object of the type `CEnemy`. When the object is initialized, it calls its appropriate constructors and the values are assigned to them. Then we call the functions by dereferencing the pointer using the `->` operator. So when we call the `p->` function, it is the same as (`*p`).function. As we are dynamically allocating memory, we should also delete the object or else we will get a memory leak. We have used `vld` to check for memory leaks. This program does not have any, as we have used the `delete` keyword. Just comment out the line `delete pEnemy;` and you will notice that the program has few memory leaks on exiting.

Using polymorphism to reuse code

Polymorphism means having several forms. Typically, we use polymorphism when there is a hierarchy of classes and they are related in some way. We generally achieve this level of relation by using inheritance.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will see how we can use the same function and override it with different functionalities based on our needs. Also, we will see how we can share values across base and derived classes:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called Source.cpp and three header files called `Enemy.h`, `Dragon.h`, and `Soldier.h`.

5. Add the following lines of code to `Enemy.h`:

```
#ifndef _ENEMY_H
#define _ENEMY_H

#include <iostream>

using namespace std;

class CEnemy {
protected:
  int m_ihealth,m_iarmourValue;
public:
  CEnemy(int ihealth, int iarmourValue) : m_ihealth(ihealth), m_
iarmourValue(iarmourValue) {}
  virtual int TotalHP(void) = 0;
  void PrintHealth()
  {
    cout << "Total health is " << this->TotalHP() << '\n';
  }
};

   #endif
```

6. Add the following lines of code to `Dragon.h`:

```
#ifndef _DRAGON_H
#define _DRAGON_H

#include "Enemy.h"
#include <iostream>

using namespace std;


class CDragon : public CEnemy {
public:
  CDragon(int m_ihealth, int m_iarmourValue) : CEnemy(m_ihealth,
m_iarmourValue)
  {
  }
  int TotalHP()
  {
    cout << "Dragon's ";
    return m_ihealth*2+3*m_iarmourValue;
  }
```

```
};

  #endif
```

7. Add the following lines of code to `Soldier.h`:

```
#ifndef _SOLDIER_H
#define _SOLDIER_H

#include "Enemy.h"
#include <iostream>

using namespace std;


class CSoldier : public CEnemy {
public:
  CSoldier(int m_ihealth, int m_iarmourValue) : CEnemy(m_ihealth,
m_iarmourValue) {}
  int TotalHP()
  {
    cout << "Soldier's ";
    return m_ihealth+m_iarmourValue;
  }
};

#endif
```

8. Add the following lines of code to `Source.cpp`:

```
// dynamic allocation and polymorphism
#include <iostream>
#include <conio.h>
#include "vld.h"
#include "Enemy.h"
#include "Dragon.h"
#include "Soldier.h"



int main()
 {
  CEnemy* penemy1 = new CDragon(100, 50);
  CEnemy* penemy2 = new CSoldier(100, 100);

  penemy1->PrintHealth();
```

```
    penemy2->PrintHealth();

    delete penemy1;
    delete penemy2;

    _getch();
    return 0;

}
```

How it works...

Polymorphism is the ability to have different forms. So in this example, we have an `Enemy` interface that does not have any functionality for calculating total health. However, we know that all types of enemy should have a function to calculate total health. So we have made the function in the base class as a pure virtual function by assigning it to `0`.

This enables, or rather forces, all the child classes to have their own implementation for calculating total health. So the `CSoldier` class and `CDragon` class have their own implementation of `TotalHP`. The advantage of such a structure is that we can create a pointer object of the child from the base and when it resolves, it calls the correct function of the child class.

If we do not create a virtual function, then the functions in the child classes would have hidden the function of the base class. With a pure virtual function, however, this is not true as this would create a compiler error. The way the compiler resolves the functions at run time is by a technique called dynamic dispatch. Most languages use dynamic dispatch. C++ uses single-cast dynamic dispatch. It does so with the help of virtual tables. When the `CEnemy` class defines the virtual function `TotalHP`, the compiler adds a hidden member variable to the class which points an array of pointers to functions called the **virtual method table** (**VMT** or **Vtable**). At runtime, these pointers will be set to point to the right function, because at compile time, it is not yet known if the base function is to be called or a derived one implemented by `CDragon` and `CSoldier`.

The member variables in the base class are protected. This means that the derived class also has access to the member variables. From the driver program, because we have allocated memory dynamically, we should also delete, or else we will have memory leaks. When the destructor is marked as virtual, we ensure that the right destructor is called.

www.allitebooks.com

Using copy constructors

Copy constructors are used to copy one object to another. C++ provides us with a default copy constructor, but it is not recommended. We should write our own copy constructor for better coding and organizing practices. It also minimizes crashes and bugs that may arise if we use the default copy constructor provided by C++.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will see how easy it is to write a copy constructor:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add source files called `Source.cpp` and `Terrain.h`.
5. Add the following lines of code to `Terrain.h`:

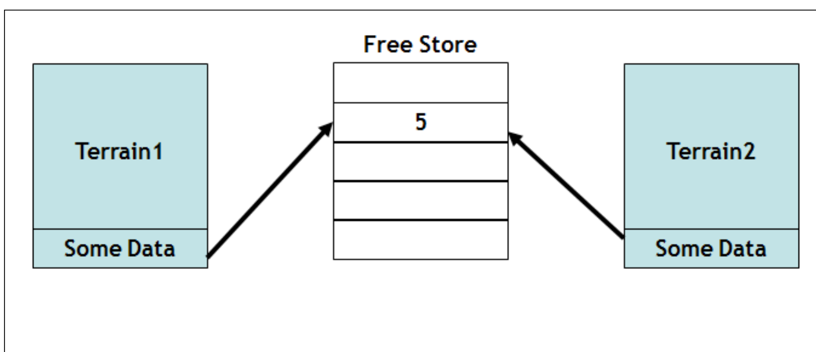
```
#pragma once
#include <iostream>

using namespace std;
class CTerrain
{
public:
  CTerrainCTerrain();
  ~CTerrain();

  CTerrain(const CTerrain &T)
  {
    cout << "\n Copy Constructor";
  }
  CTerrain& operator =(const CTerrain &T)
  {
    cout << "\n Assignment Operator";
    return *this;
  }
};
```

6. Add the following lines of code to `Source.cpp`:

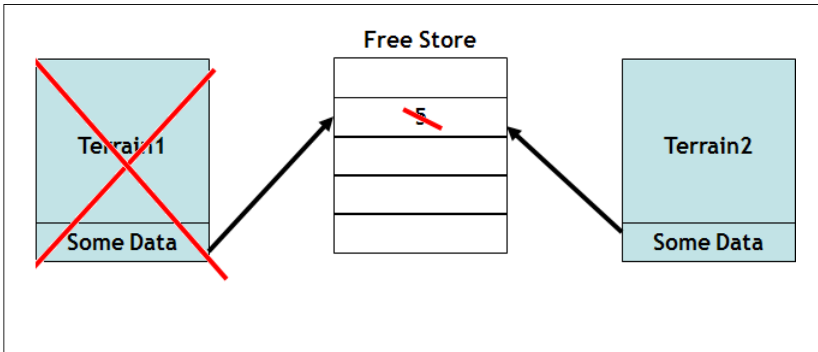
```
#include <conio.h>
#include "Terrain.h"

using namespace std;

int main()
{
  CTerrain Terrain1,Terrain2;

  Terrain1 = Terrain2;

  CTerrain Terrain3 = Terrain1;

  _getch();
  return 0;
}
```

How it works...

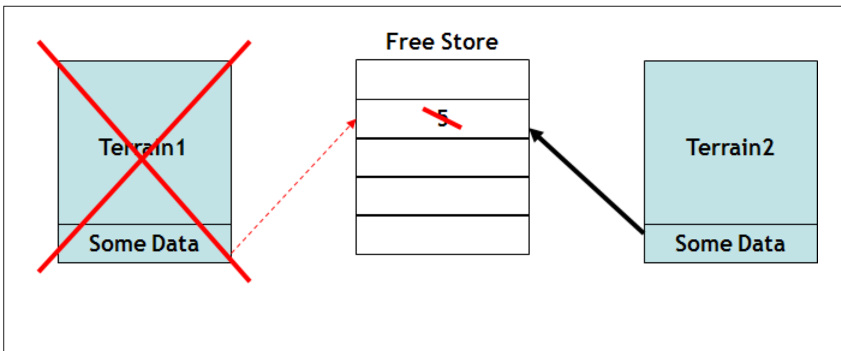
In this example, we have created our own copy constructor and an assignment operator. When we assign two objects that are already initialized, then the assignment operator is called. When we initialize an object and set it to the other object, a copy constructor is called. If we do not create our own copy constructor, the newly created object just holds a shallow reference of the object it is being assigned to. If the object gets destroyed, then the shallow object becomes lost as the memory is also lost. If we create our own copy constructor, a deep copy is created and even if the first object is deleted, the second object stills holds the information in a different memory location.



So in effect, a shallow copy (or member-wise copy) copies the exact values of one object's member variables into another object. Pointers in both objects end up pointing to the same memory. A deep copy copies the values allocated on the free store to newly allocated memory. So in shallow deleting, the object in the shallow copy is disastrous:



However, a deep copy solves this problem for us:



Use operator overloading to reuse operators

There are lots of operators that are provided for us by C++. However, sometimes we need to overload these operators so that we can use them on data structures that we create ourselves. Of course, we can overload the operators to change the meaning as well. For example, we can change + (plus) to behave like - (minus), but this is not recommended as this usually does not serve any purpose or help us in any way. Also, it may confuse other programmers who are using the same code base.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will see how we can overload an operator and which operators are allowed to be overloaded in C++.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`, `vector3.h`, and `vector3.cpp`.
5. Add the following lines of code to `Source.cpp`:

```
#include "vector3.h"
#include <conio.h>
#include "vld.h"

int main()
{
  // Vector tests:

  // Create two vectors.
  CVector3 a(1.0f, 2.0f, 3.0f);
  CVector3 b(1.0f, 2.0f, 3.0f);

  CVector3 c;

  // Zero Vector.
  c.Zero();

  // Addition.
  CVector3 d = a + b;

  // Subtraction.
  CVector3 e = a - b;

  //Scalar Multiplication.
  CVector3 f1 = a * 10;

  //Scalar Multiplication.
  CVector3 f2 = 10 * a;

  //Scalar Division.
```

```
CVector3 g = a / 10;

// Unary minus.
CVector3 h = -a;

// Relational Operators.
bool bAEqualsB = (a == b);
bool bANotEqualsB = (a != b);

// Combined operations +=.
c = a;
c += a;

// Combined operations -=.
c = a;
c -= a;

// Combined operations /=.
c = a;
c /= 10;

// Combined operations *=.
c = a;
c *= 10;

// Normalization.
c.Normalize();

// Dot Product.
float fADotB = a * b;

// Magnitude.
float fMag1 = CVector3::Magnitude(a);
float fMag2 = CVector3::Magnitude(c);

// Cross product.
CVector3 crossProduct = CVector3::CrossProduct(a, c);

// Distance.
float distance = CVector3::Distance(a, c);

_getch();
```

```
    return (0);


}
```

6. Add the following lines of code to `vector3.h`:

```
#ifndef __VECTOR3_H__
#define __VECTOR3_H__

#include <cmath>

class CVector3
{
public:
  // Public representation: Not many options here.
  float x;
  float y;
  float z;

  CVector3();
  CVector3(const CVector3& _kr);
  CVector3(float _fx, float _fy, float _fz);

  // Assignment operator.
  CVector3& operator =(const CVector3& _kr);

  // Relational operators.
  bool operator ==(const CVector3& _kr) const;
  bool operator !=(const CVector3& _kr) const;

  // Vector operations
  void Zero();

  CVector3 operator -() const;
  CVector3 operator +(const CVector3& _kr) const;
  CVector3 operator -(const CVector3& _kr) const;

  // Multiplication and division by scalar.
  CVector3 operator *(float _f) const;
  CVector3 operator /(float _f) const;

  // Combined assignment operators to conform to C notation
convention.
```

```
    CVector3& operator +=(const CVector3& _kr);
    CVector3& operator -=(const CVector3& _kr);
    CVector3& operator *=(float _f);
    CVector3& operator /=(float _f);

    // Normalize the vector
    void Normalize();
    // Vector dot product.
    // We overload the standard multiplication symbol to do this.
    float operator *(const CVector3& _kr) const;

    // Static member functions.

    // Compute the magnitude of a vector.
    static inline float Magnitude(const CVector3& _kr)
    {
      return (sqrt(_kr.x * _kr.x + _kr.y * _kr.y + _kr.z * _kr.z));
    }

    // Compute the cross product of two vectors.
    static inline CVector3 CrossProduct(const CVector3& _krA,
      const CVector3& _krB)
    {
      return
        (
        CVector3(_krA.y * _krB.z - _krA.z * _krB.y,
        _krA.z * _krB.x - _krA.x * _krB.z,
        _krA.x * _krB.y - _krA.y * _krB.x)
        );
    }

    // Compute the distance between two points.
    static inline float Distance(const CVector3& _krA, const
  CVector3& _krB)
    {
      float fdx = _krA.x - _krB.x;
      float fdy = _krA.y - _krB.y;
      float fdz = _krA.z - _krB.z;

      return sqrt(fdx * fdx + fdy * fdy + fdz * fdz);
    }
  };

  // Scalar on the left multiplication, for symmetry.
```

```
inline CVector3 operator *(float _f, const CVector3& _kr)
{
  return (CVector3(_f * _kr.x, _f * _kr.y, _f * _kr.z));
}

#endif // __VECTOR3_H__
```

7. Add the following lines of code to `vector3.cpp`:

```
#include "vector3.h"

// Default constructor leaves vector in an indeterminate state.
CVector3::CVector3()
{

}

// Copy constructor.
CVector3::CVector3(const CVector3& _kr)
: x(_kr.x)
, y(_kr.y)
, z(_kr.z)
{

}

// Construct given three values.
CVector3::CVector3(float _fx, float _fy, float _fz)
: x(_fx)
, y(_fy)
, z(_fz)
{

}

// Assignment operator, we adhere to C convention and return
reference to the lvalue.
CVector3&
CVector3::operator =(const CVector3& _kr)
{
  x = _kr.x;
  y = _kr.y;
  z = _kr.z;

  return (*this);
```

```
}

// Equality operator.
bool
CVector3::operator ==(const CVector3&_kr) const
{
  return (x == _kr.x && y == _kr.y && z == _kr.z);
}

// Inequality operator.
bool
CVector3::operator !=(const CVector3& _kr) const
{
  return (x != _kr.x || y != _kr.y || z != _kr.z);
}


// Set the vector to zero.
void
CVector3::Zero()
{
  x = 0.0f;
  y = 0.0f;
  z = 0.0f;
}

// Unary minus returns the negative of the vector.
CVector3
CVector3::operator -() const
{
  return (CVector3(-x, -y, -z));
}

// Binary +, add vectors.
CVector3
CVector3::operator +(const CVector3& _kr) const
{
  return (CVector3(x + _kr.x, y + _kr.y, z + _kr.z));
}

// Binary -, subtract vectors.
CVector3
CVector3::operator -(const CVector3& _kr) const
{
```

```
  return (CVector3(x - _kr.x, y - _kr.y, z - _kr.z));
}

// Multiplication by scalar.
CVector3
CVector3::operator *(float _f) const
{
  return (CVector3(x * _f, y * _f, z * _f));
}

// Division by scalar.
// Precondition: _f must not be zero.
CVector3
CVector3::operator /(float _f) const
{
  // Warning: no check for divide by zero here.
  ASSERT(float fOneOverA = 1.0f / _f);

  return (CVector3(x * fOneOverA, y * fOneOverA, z * fOneOverA));
}

CVector3&
CVector3::operator +=(const CVector3& _kr)
{
  x += _kr.x;
  y += _kr.y;
  z += _kr.z;

  return (*this);
}

CVector3&
CVector3::operator -=(const CVector3& _kr)
{
  x -= _kr.x;
  y -= _kr.y;
  z -= _kr.z;

  return (*this);
}

CVector3&
CVector3::operator *=(float _f)
{
```

```
    x *= _f;
    y *= _f;
    z *= _f;

    return (*this);
  }

  CVector3&
  CVector3::operator /=(float _f)
  {
    float fOneOverA = ASSERT(1.0f / _f);

    x *= fOneOverA;
    y *= fOneOverA;
    z *= fOneOverA;

    return (*this);
  }

  void
  CVector3::Normalize()
  {
    float fMagSq = x * x + y * y + z * z;

    if (fMagSq > 0.0f)
    {
      // Check for divide-by-zero.
      float fOneOverMag = 1.0f / sqrt(fMagSq);

      x *= fOneOverMag;
      y *= fOneOverMag;
      z *= fOneOverMag;
    }
  }

  // Vector dot product.
  //    We overload the standard multiplication symbol to do this.
  float
  CVector3::operator *(const CVector3& _kr) const
  {
    return (x * _kr.x + y * _kr.y + z * _kr.z);
  }
```

How it works...

C++ has built-in types: int, char, and float. Each of these types has a number of built-in operators, such as addition (+) and multiplication (*). C++ allows you to add these operators to your own classes as well. Operators on built-in types (int, float) cannot be overloaded. The precedence order cannot be changed. There are many reasons for proceeding with caution when overloading an operator. The goal is to increase usability and understanding. In our example, we have overloaded the basic multiplication operators so that we can add, subtract, and so on our `vector3` objects that we create. This is extremely handy, as we can find the distance of an object in our game if we know the position vectors of the two objects. We have used const functions as much as possible. The compiler will enforce the promise to not modify the object. This can be a great way to make sure that your code has no unanticipated side effects.

All functions that accept vectors accept a constant reference to a vector. We have to remember that passing an argument by value to a function invokes a constructor. Inheritance will not be very useful to the vector class, as we know `CVector3` is speed critical. The V-table adds 25% to the class size, so it is not advisable.

Also, data hiding does not make too much sense, as we need the values of the vector class. Some operators can be overloaded in C++. The operators that C++ does not allow us to overload are:

```
(Member Access or Dot operator),?: (Ternary or Conditional
Operator),:: (Scope Resolution Operator),.* (Pointer-to-member
Operator),sizeof (Object size Operator) and typeid (Object type
Operator)
```

Use function overloading to reuse functions

Function overloading is an important concept in C++. Sometimes, we want to use the same function name but have different functions to work on different data types or a different number of types. This is useful as the client can choose the correct function based on its needs. C++ allows us to do this by using function overloading.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will learn how to overload a function:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a **Win32 Console Application**.
4. Add source files called `main.cpp`, `Cspeed.h`, and `Cspeed.cpp`.
5. Add the following lines of code to `main.cpp`:

```
#include <iostream>
#include <conio.h>
#include "CSpeed.h"

using namespace std;



//This is not overloading as the function differs only
//in return type
/*int Add(float x, float y)
{
  return x + y;
}*/

int main()
{
  CSpeed speed;

  cout<<speed.AddSpeed(2.4f, 7.9f)<<endl;
  cout << speed.AddSpeed(4, 5)<<endl;
  cout << speed.AddSpeed(4, 9, 12)<<endl;



  _getch();
  return 0;
}
```

6. Add the following lines of code to `CSpeed.cpp`:

```
#include "CSpeed.h"


CSpeed::CSpeed()
{

}

CSpeed::~CSpeed()
{

}
int CSpeed::AddSpeed(int x, int y, int z)
{
  return x + y + z;
}
int CSpeed::AddSpeed(int x, int y)
{
  return x + y;
}
float CSpeed::AddSpeed(float x, float y)
{
  return x + y;
}
```

7. Add the following lines of code to `CSpeed.h`:

```
#ifndef _VELOCITY_H
#define _VELOCITY_H

class CSpeed
{
public:
  int AddSpeed(int x, int y, int z);
  int AddSpeed(int x, int y);
  float AddSpeed(float x, float y);

  CSpeed();
  ~CSpeed();
```

```
private:


};

#endif
```

How it works...

Overloading a function is a type of functional polymorphism. A function can be overloaded only by the number of parameters in the argument list and the type of parameter. A function cannot be overloaded only by the return type.

We have created a class to calculate the sum of speeds. We can use the function to add two speeds, three speeds, or speeds of different data types. The compiler will resolve which function to call based on the signature. One might argue that we could create different objects with different speeds and then add them using operator overloading, or use templates and write one template function. However, we have to remember that in simple templates the implementation will remain the same, but in function overloading we can change the implementation of each function as well.

Using files for input and output

Files are really useful for saving data locally, so we can retrieve it the next time the program is run or analyze the data after the program exits. For all data structures that we create in code and populate with values, the values will get lost after the application quits unless we save them locally or to the server/cloud. Files serve the purpose of containing the saved data. We can create text files, binary files, or even a file with our own encryption. Files are very handy when we want to log errors or generate a crash report.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how to use file handling operations in C++ to write or read from a text file. We can even use C++ operations to create binary files.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.

4. Add source files called `Source.cpp`, `File.h`, and `File.cpp`.
5. Add the following lines of code to `Source.cpp`:

```
#include <conio.h>
#include "File.h"

int main() {

  CFile file;

  file.WriteNewFile("Example.txt");
  file.WriteNewFile("Example.txt", "Logging text1");
  file.AppendFile("Example.txt", "Logging text2");
  file.ReadFile("Example.txt");

  _getch();
  return 0;
}
```

6. Add the following lines of code to `File.cpp`:

```
#include "File.h"
#include <string>
#include <fstream>
#include <iostream>

using namespace std;

CFile::CFile()
{
  Text = "This is the initial data";
}
CFile::~CFile()
{

}
void CFile::WriteNewFile(string Filename)const
{
  ofstream myfile(Filename);
  if (myfile.is_open())
  {
    myfile << Text;

    myfile.close();
  }
```

```
  else cout << "Unable to open file";
}
void CFile::WriteNewFile(string Filename,string Text)const
{
  ofstream myfile(Filename);
  if (myfile.is_open())
  {
    myfile << Text;

    myfile.close();
  }
  else cout << "Unable to open file";
}

void CFile::AppendFile(string Filename, string Text)const
{
  ofstream outfile;

  outfile.open(Filename, ios_base::app);
  outfile << Text;
       outfile.close();

}
void CFile::ReadFile(string Filename)const
{
  string line;
  ifstream myfile(Filename);
  if (myfile.is_open())
  {
    while (getline(myfile, line))
    {
      cout << line << '\n';
    }
    myfile.close();
  }

  else cout << "Unable to open file";
}
```

7. Add the following lines of code to `File.h`:

```
#ifndef _FILE_H
#define _FILE_H

#include <iostream>
```

```
#include <string.h>
using namespace std;

class CFile
{
public:
  CFile();
  ~CFile();

  void WriteNewFile(string Filename)const;
  void WriteNewFile(string Filename, string Text)const;
  void AppendFile(string Filename, string Text)const;
  void ReadFile(string Filename)const;
private:

  string Text;
};
#endif
```

How it works...

We use file handling for a variety of reasons. Some of the most important reasons are to log data while the game is running, to load data from a text file to be used in the game, or to encrypt the save data or load data of a game.

We have created a class called `CFile`. This class helps us to write data to a new file, to append to a file, and to read from a file. We use the `fstream` header file to load all the file handling operations.

Everything in a file is written and read in terms of streams. While doing C++ programming, we must write information to a file from our program using the stream insertion operator (`<<`), just as we use that operator to output information to the screen. The only difference is that you use an `ofstream` or `fstream` object, instead of the `cout` object.

We have created a constructor to contain initial data if a file is created without any data in it. If we just create or write to a file, each time a new file will be created with the new data. This is sometimes useful if we just want to write the most recently updated or latest data. However, if we want to add data to an existing file, we can use the `append` function. The append function starts writing to an existing file from the last file-position pointer position.

The read function starts reading data from the file until it reaches the last line of written data. We can display the result to the screen or, if needed, we could then write the contents to another file. We also must remember to close the file after each operation, or it might lead to ambiguity in the code. We can also use the `seekp` and `seekg` functions to reposition the file-position pointer.

Creating your first simple game

Creating a simple text-based game is really easy. All we need to do is to create some rules and logic and we will have ourselves a game. Of course, as the game gets more complex we need to add more functions. When the game reaches a point where there are multiple behaviors and states of objects and enemies, we should use classes and inheritance to achieve the desired result.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will learn how to create a simple luck-based lottery game:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a `Source.cpp` file.
5. Add the following lines of code to it:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main(void) {
  srand(time(NULL)); // To not have the same numbers over and over
again.

  while (true) { // Main loop.
    // Initialize and allocate.
    int inumber = rand() % 100 + 1 // System number is stored in
here.
    int iguess; // User guess is stored in here.
    int itries = 0; // Number of tries is stored here.
    char canswer; // User answer to question is stored here.



    while (true) { // Get user number loop.
```

```
      // Get number.
      std::cout << "Enter a number between 1 and 100 (" << 20 -
itries << " tries left): ";
      std::cin >> iguess;
      std::cin.ignore();

      // Check is tries are taken up.
      if (itries >= 20) {
        break;
      }

      // Check number.
      if (iguess > inumber) {
        std::cout << "Too high! Try again.\n";
      }
      else if (iguess < inumber) {
        std::cout << "Too low! Try again.\n";
      }
      else {
        break;
      }

      // If not number, increment tries.
      itries++;
    }

    // Check for tries.
    if (itries >= 20) {
      std::cout << "You ran out of tries!\n\n";
    }
    else {
      // Or, user won.
      std::cout << "Congratulations!! " << std::endl;
      std::cout << "You got the right number in " << itries << "
tries!\n";
    }

    while (true) { // Loop to ask user is he/she would like to
play again.
      // Get user response.
      std::cout << "Would you like to play again (Y/N)? ";
      std::cin >> canswer;
```

```
            std::cin.ignore();

            // Check if proper response.
            if (canswer == 'n' || canswer == 'N' || canswer == 'y' ||
      canswer == 'Y') {
              break;
            }
            else {
              std::cout << "Please enter \'Y\' or \'N\'...\n";
            }
          }

          // Check user's input and run again or exit;
          if (canswer == 'n' || canswer == 'N') {
            std::cout << "Thank you for playing!";
            break;
          }
          else {
            std::cout << "\n\n\n";
          }
        }

        // Safely exit.
        std::cout << "\n\nEnter anything to exit. . . ";
        std::cin.ignore();
        return 0;
      }
```

How it works...

The game works by creating a random number from 1 to 100 and asks the user to guess that number. Hints are provided as to whether the number guessed is higher or lower than the actual number. The user is given just 20 tries to guess the number. We first need a pseudo seeder, based on which we are going to generate a random number. The pseudo seeder in this case is `srand`. We have chosen TIME as a value to generate our random range.

We need to execute the program in an infinite loop so that the program breaks only when all tries are used up or when the user correctly guesses the number. We can set a variable for tries and increment for every guess a user takes. The random number is generated by the rand function. We use `rand%100+1` so that the random number is in the range 1 to 100. We ask the user to input the guessed number and then we check whether that number is less than, greater than, or equal to the randomly generated number. We then display the correct message. If the user has guessed correctly, or all tries have been used, the program should break out of the main loop. At this point, we ask the user whether they want to play the game again.

Then, depending on the answer, we go back into the main loop and start the process of selecting a random number.

Templates – when to use them

Templates are a C++ programming way to lay the foundations for writing a generic program. Using templates, we can write code in such a way that it is independent of any particular data type. We can use function templates or class templates.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out the importance of templates, how to use them, and the advantages that using them provides us.

1. Open Visual Studio.
2. Create a new C++ project.
3. Add source files called `Source.cpp` and `Stack.h`.
4. Add the following lines of code to `Source.cpp`:

```
#include <iostream>
#include <conio.h>
#include <string>
#include "Stack.h"

using namespace std;

template<class T>
void Print(T array[], int array_size)
{
  for (int nIndex = 0; nIndex < array_size; ++nIndex)
  {
    cout << array[nIndex] << "\t";
  }
  cout << endl;
}

int main()
{
  int iArray[5] = { 4, 5, 6, 6, 7 };
```

```
    char cArray[3] = { 's', 's', 'b' };
    string sArray[3] = { "Kratos", "Dr.Evil", "Mario" };

    //Printing any type of elements
    Print(iArray, sizeof(iArray) / sizeof(*iArray));
    Print(cArray, sizeof(cArray) / sizeof(*cArray));
    Print(sArray, sizeof(sArray) / sizeof(*sArray));


    Stack<int> iStack;

    //Pushes an element to the bottom of the stack
    iStack.push(7);

    cout << iStack.top() << endl;

    for (int i = 0; i < 10; i++)
    {
      iStack.push(i);
    }

    //Removes an element from the top of the stack
    iStack.pop();

    //Prints the top of stack
    cout << iStack.top() << endl;

    _getch();
  }
```

5. Add the following lines of code to `Stack.h`:

```
#include <vector>


using namespace std;

template <class T>
class Stack {
private:
  vector<T> elements;     // elements

public:
  void push(T const&);  // push element
  void pop();               // pop element
```

```
  T top() const;             // return top element
  bool empty() const{        // return true if empty.
    return elements.empty();
  }
};

template <class T>
void Stack<T>::push(T const& elem)
{
  // append copy of passed element
  elements.push_back(elem);
}

template <class T>
void Stack<T>::pop()
{
  if (elements.empty()) {
    throw out_of_range("Stack<>::pop(): empty stack");
  }
  // remove last element
  elements.pop_back();
}

template <class T>
T Stack<T>::top() const
{
  if (elements.empty()) {
    throw out_of_range("Stack<>::top(): empty stack");
  }
  // return copy of last element
  return elements.back();
}
```

How it works...

Templates are the foundation of generic programming in C++. If the implementation of a function or a class is the same but we need them to operate on different data types, it is advisable to use templates instead of writing a new class or function. One can argue that we can overload a function to achieve the same thing, but keep in mind that while overloading a function, we can change the implementation based on the data type and we are still writing a new function. With templates, the implementation has to be the same for all data types. This is the advantage of templates: writing one function is enough. With advanced templates and C++11 features, we can even change the implementation, but we will reserve that discussion for later.

We have used function templates and class templates in this example. The function template is defined in `Source.cpp` itself. On top of the `print` function, we have added the line template `<class T>`. The `keyword` class could be replaced by `typename` as well. The reason for two keywords is a historic one and we do not need to discuss it here. The remaining part of the function definition is normal, except instead of using a particular data type, we have used `T`. So when we call the function from main, `T` gets replaced with the correct data type. In this way, by just using one function, we can print all data types. We can even create our own data type and pass it to the function.

`Stack.h` is an example of a class template, as the data type that the class uses is a generic one. We have selected a stack as it is a very popular data structure in games programming. It's a **LIFO** (**Last In First Out**) structure, so we can display the latest content from the stack as per our requirements. The push function pushes an element onto the stack, whereas a pop removes an element from the stack. The top function displays the top-most element of the stack and the empty function empties the stack. By using this generic stack class, we can store and display the data type of our choice.

One thing to be kept in mind while using templates is that the compiler must know at compile time the correct implementation of the template, so generally template definition and declaration are both done in the header file. However, if you want to separate out the two, you can do so with two popular methods. One method is to have another header file and list the implementation at the end of it. The other implementation is to create an `.ipp` or `.tpp` file extension and have the implementation in those files.

3
Data Structures in Game Development

In chapter, the following recipes will be covered:

- Using more advanced data structures
- Using linked lists to store data
- Using stacks to store data
- Using queues to store data
- Using trees to store data
- Using graphs to store data
- Using STL lists to store data
- Using STL maps to store data
- Using STL hash tables to store data

Introduction

Data structures are used in the video games industry to organize code into more cleaner and more manageable. An average video game will have about 20,000 lines of code at least. If we do not use an effective storage system and structure to manage that code, it will become very difficult to debug. Also, we may end up writing the same code multiple times.

Data structures are also very useful for searching elements if we have a large data set. Let us consider that we are making a MMO. From the thousands of players online playing the game, we need to isolate a player who has the most points on a certain day. If we have not organized the user data into some meaningful data structure, this might take a long time. On the other hand, using a suitable data structure can help us achieve this within seconds.

Using more advanced data structures

In this recipe, we will see how to use advanced data structures. The main task of a programmer is to choose the correct data structure based on the need, so that the time taken to store and parse the data is minimized. Sometimes the choice of a correct data structure is more important than selecting a suitable algorithm to solve a problem.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to use advanced data structures and why we should use them. If we organize data into suitable structures, it becomes faster to access data and easier to apply complex algorithms to it.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add source files called `Source.cpp`, `LinkedList.h`/ `LinkedList.cpp` and `HashTables.h`/ `HashTables.cpp`.
5. Add the following lines of code to `Source.cpp`:

```
#include "HashTable.h"
#include <conio.h>

int main()
{
  // Create 26 Items to store in the Hash Table.
  Item * A = new Item{ "Enemy1", NULL };
  Item * B = new Item{ "Enemy2", NULL };
  Item * C = new Item{ "Enemy3", NULL };
  Item * D = new Item{ "Enemy4", NULL };
  Item * E = new Item{ "Enemy5", NULL };
  Item * F = new Item{ "Enemy6", NULL };
  Item * G = new Item{ "Enemy7", NULL };
  Item * H = new Item{ "Enemy8", NULL };
  Item * I = new Item{ "Enemy9", NULL };
  Item * J = new Item{ "Enemy10", NULL };
```

```
Item * K = new Item{ "Enemy11", NULL };
Item * L = new Item{ "Enemy12", NULL };
Item * M = new Item{ "Enemy13", NULL };
Item * N = new Item{ "Enemy14", NULL };
Item * O = new Item{ "Enemy15", NULL };
Item * P = new Item{ "Enemy16", NULL };
Item * Q = new Item{ "Enemy17", NULL };
Item * R = new Item{ "Enemy18", NULL };
Item * S = new Item{ "Enemy19", NULL };
Item * T = new Item{ "Enemy20", NULL };
Item * U = new Item{ "Enemy21", NULL };
Item * V = new Item{ "Enemy22", NULL };
Item * W = new Item{ "Enemy23", NULL };
Item * X = new Item{ "Enemy24", NULL };
Item * Y = new Item{ "Enemy25", NULL };
Item * Z = new Item{ "Enemy26", NULL };

// Create a Hash Table of 13 Linked List elements.
HashTable table;

// Add 3 Items to Hash Table.
table.insertItem(A);
table.insertItem(B);
table.insertItem(C);
table.printTable();


// Remove one item from Hash Table.
table.removeItem("Enemy3");
table.printTable();



// Add 23 items to Hash Table.
table.insertItem(D);
table.insertItem(E);
table.insertItem(F);
table.insertItem(G);
table.insertItem(H);
table.insertItem(I);
table.insertItem(J);
table.insertItem(K);
table.insertItem(L);
table.insertItem(M);
```

```
  table.insertItem(N);
  table.insertItem(O);
  table.insertItem(P);
  table.insertItem(Q);
  table.insertItem(R);
  table.insertItem(S);
  table.insertItem(T);
  table.insertItem(U);
  table.insertItem(V);
  table.insertItem(W);
  table.insertItem(X);
  table.insertItem(Y);
  table.insertItem(Z);
  table.printTable();


  // Look up an item in the hash table
  Item * result = table.getItemByKey("Enemy4");
  if (result!=nullptr)
  cout << endl<<"The next key is "<<result->next->key << endl;

  _getch();
  return 0;
}
```

6. Add the following lines of code to `LinkedList.h`:

```
#ifndef LinkedList_h
#define LinkedList_h

#include <iostream>
#include <string>
using namespace std;

//*****************************************************************
*
// List items are keys with pointers to the next item.
//*****************************************************************
*
struct Item
{
  string key;
```

```
  Item * next;
};

//******************************************************************
*
// Linked lists store a variable number of items.
//******************************************************************
*
class LinkedList
{
private:
  // Head is a reference to a list of data nodes.
  Item * head;

  // Length is the number of data nodes.
  int length;

public:
  // Constructs the empty linked list object.
  // Creates the head node and sets length to zero.
  LinkedList();

  // Inserts an item at the end of the list.
  void insertItem(Item * newItem);

  // Removes an item from the list by item key.
  // Returns true if the operation is successful.
  bool removeItem(string itemKey);

  // Searches for an item by its key.
  // Returns a reference to first match.
  // Returns a NULL pointer if no match is found.
  Item * getItem(string itemKey);

  // Displays list contents to the console window.
  void printList();

  // Returns the length of the list.
  int getLength();

  // De-allocates list memory when the program terminates.
  ~LinkedList();
};

#endif
```

7. Add the following lines of code to `LinkedList.cpp`:

```
#include "LinkedList.h"

// Constructs the empty linked list object.
// Creates the head node and sets length to zero.
LinkedList::LinkedList()
{
  head = new Item;
  head->next = NULL;
  length = 0;
}

// Inserts an item at the end of the list.
void LinkedList::insertItem(Item * newItem)
{
  if (!head->next)
  {
    head->next = newItem;
newItem->next=NULL;
    length++;
    return;
  }
//Can be reduced to fewer lines of codes.
//Using 2 variables p and q to make it more clear
  Item * p = head->next;
  Item * q = p->next;
  while (q)
  {
    p = q;
    q = p->next;
  }
  p->next = newItem;
  newItem->next = NULL;
  length++;
}

// Removes an item from the list by item key.
// Returns true if the operation is successful.
bool LinkedList::removeItem(string itemKey)
{
  if (!head->next) return false;
  Item * p = head;
  Item * q = head->next;
  while (q)
```

```
  {
    if (q->key == itemKey)
    {
      p->next = q->next;
      delete q;
      length--;
      return true;
    }
    p = q;
    q = p->next;
  }
  return false;
}

// Searches for an item by its key.
// Returns a reference to first match.
// Returns a NULL pointer if no match is found.
Item * LinkedList::getItem(string itemKey)
{
  Item * p = head;
  Item * q = p->next;
  while (q)
  {

if (q->key == itemKey))
  {
return p;
  }
p = q;
q = p->next;
  }
  return NULL;
}

// Displays list contents to the console window.
void LinkedList::printList()
{
  if (length == 0)
  {
    cout << "\n{ }\n";
    return;
  }
  Item * p = head;
```

```
    Item * q = p->next;
    cout << "\n{ ";
    while (q)
    {
      p = q;
      if (p != head)
      {
        cout << p->key;
        if (q->next) cout << ", ";
        else cout << " ";
      }
      q = p->next;
    }
    cout << "}\n";
  }

  // Returns the length of the list.
  int LinkedList::getLength()
  {
    return length;
  }

  // De-allocates list memory when the program terminates.
  LinkedList::~LinkedList()
  {
    Item * p = head;
    Item * q = head;
    while (q)
    {
      p = q;
      q = p->next;
      if (q)
    }
  delete p;
  }
```

8. Add the following lines of code to `HashTable.cpp`:

```
#include "HashTable.h"

// Constructs the empty Hash Table object.
// Array length is set to 13 by default.
HashTable::HashTable(int tableLength)
{
  if (tableLength <= 0) tableLength = 13;
```

```
    array = new LinkedList[tableLength];
    length = tableLength;
  }

  // Returns an array location for a given item key.
  int HashTable::hash(string itemKey)
  {
    int value = 0;
    for (int i = 0; i < itemKey.length(); i++)
      value += itemKey[i];
    return (value * itemKey.length()) % length;
  }

  // Adds an item to the Hash Table.
  void HashTable::insertItem(Item * newItem)
  {
  If(newItem)
  {
    int index = hash(newItem->key);
    array[index].insertItem(newItem);
  }
  }

  // Deletes an Item by key from the Hash Table.
  // Returns true if the operation is successful.
  bool HashTable::removeItem(string itemKey)
  {
    int index = hash(itemKey);
    return array[index].removeItem(itemKey);
  }

  // Returns an item from the Hash Table by key.
  // If the item isn't found, a null pointer is returned.
  Item * HashTable::getItemByKey(string itemKey)
  {
    int index = hash(itemKey);
    return array[index].getItem(itemKey);
  }

  // Display the contents of the Hash Table to console window.
  void HashTable::printTable()
  {
    cout << "\n\nHash Table:\n";
    for (int i = 0; i < length; i++)
```

```
    {
      cout << "Bucket " << i + 1 << ": ";
      array[i].printList();
    }
  }


// Returns the number of locations in the Hash Table.
int HashTable::getLength()
{
  return length;
}

// Returns the number of Items in the Hash Table.
int HashTable::getNumberOfItems()
{
  int itemCount = 0;
  for (int i = 0; i < length; i++)
  {
    itemCount += array[i].getLength();
  }
  return itemCount;
}

// De-allocates all memory used for the Hash Table.
HashTable::~HashTable()
{
  delete[] array;
}
```

9. Add the following lines of code to `HashTables.h`:

```
#ifndef HashTable_h
#define HashTable_h

#include "LinkedList.h"

//******************************************************************
*
// Hash Table objects store a fixed number of Linked Lists.
//******************************************************************
*
class HashTable
{
```

```
private:

  // Array is a reference to an array of Linked Lists.
  LinkedList * array;

  // Length is the size of the Hash Table array.
  int length;

  // Returns an array location for a given item key.
  int hash(string itemKey);

public:

  // Constructs the empty Hash Table object.
  // Array length is set to 13 by default.
  HashTable(int tableLength = 13);

  // Adds an item to the Hash Table.
  void insertItem(Item * newItem);

  // Deletes an Item by key from the Hash Table.
  // Returns true if the operation is successful.
  bool removeItem(string itemKey);

  // Returns an item from the Hash Table by key.
  // If the item isn't found, a null pointer is returned.
  Item * getItemByKey(string itemKey);

  // Display the contents of the Hash Table to console window.
  void printTable();

  // Returns the number of locations in the Hash Table.
  int getLength();

  // Returns the number of Items in the Hash Table.
  int getNumberOfItems();

  // De-allocates all memory used for the Hash Table.
  ~HashTable();
};

#endif
```

How it works...

We have created this class to store different enemies using a hash table and then search for a particular enemy from the hash table using a key. The hash table in turn is created using a linked list.

In the `LINKEDLIST` file, we have defined a struct to store a key and a pointer to the next value in the hash table. The main class contains a pointer reference of the struct called `ITEM`. Apart from that, the class contains length of the data and member functions to insert an item, remove an item, find an element, display the entire list, and find the length of the list.

In the `HASHTABLE` file, a hash table is created using a linked list. A reference of the linked list is created along with the length of the hash table array and a private function to return an array location of a particular item in the hash table array. Apart from that, the hash table has similar functionalities to the linked list, such as inserting an item, removing an item, and displaying the hash table.

From the driver program, an object of the struct is created to initialize the items to be pushed into the hash table. Then an object of the hash table is created and the items are pushed to the table and displayed. An item is also deleted from the table. Finally, a particular item called `Enemy4` is searched and the next key is displayed.

Using linked lists to store data

In this recipe, we will see how we can use linked lists to store and organize data. The main advantage of a linked list in the games industry is that it is a dynamic data structure. However, it is bad for searching and inserting elements, as you need to find the information. The search is $O(n)$. This means we can assign memory to this data structure at runtime. In games, most things are created, destroyed, and updated at runtime, so using a linked list is very suitable. Linked lists can also be used to create linear data structures such as stacks and queues, which are equally important in game programming.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

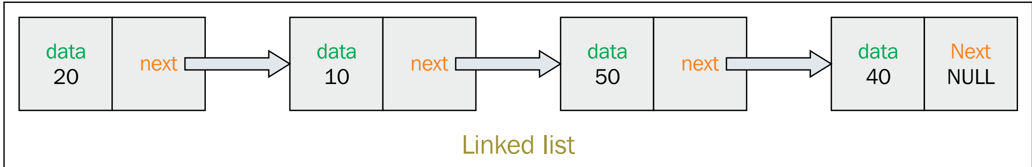
In this recipe, we will see how easy it is to use linked lists. Linked lists are a great way to store data and are used as base mechanics for other data structures:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;

typedef struct LinkedList {
  int LevelNumber;
  LinkedList * next;
} LinkedList;

int main() {
  LinkedList * head = NULL;
  int i;
  for (i = 1; i <= 10; i++) {
    LinkedList * currentNode = new LinkedList;
    currentNode->LevelNumber = i;
    currentNode->next = head;
    head = currentNode;
  }
  while (head) {
    cout << head->LevelNumber << " ";
    head = head->next;
  }
delete head;
  _getch();
  return 0;
}
```

How it works...

A linked list is used to create a data structure that stores data, and a field that contains the address of the next node. A linked list is made up of nodes.



In our example, we have created a linked list using struct and used an iteration to populate the linked list. The main concept of a linked list, as explained before, is that it contains data of some kind and contains the address information of the next node. In our example, we have created a linked list to store the current level number and the address of the next level to be loaded. This kind of structure is really important in order to store the levels we want to load. Just by traversing the linked list, we can load the levels in the correct order. Even checkpoints in the game can be programmed in a similar manner.

Using stacks to store data

Stacks are an example of a linear data structure in C++. In this type of data structure, the order in which the data is entered into the data structure is very important. The last piece of data to be entered is the first piece of data to be deleted. That is why this is sometimes also referred to as the **last in first out** (**LIFO**) data structure. The process for entering data into a stack is called **push**, and the process of deleting data is called **pop**. Sometimes we just want to print the value at the top of the stack, without deleting or popping. The stack is used in a variety of areas in the games industry, but especially when creating a UI system for a game.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how easy it is to use the stack data structure. A stack is one of the easiest data structures to implement and is used in multiple areas:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.

4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>
#include <string>

using namespace std;


class Stack
{
private:
  string UI_Elements[10];
  int top;
public:
  Stack()
  {
    top = -1;
  }

  void Push(string element)
  {
    if (top >= 10)
    {
      cout << "Some error occurred";
    }
    UI_Elements[++top] = element;
  }

  string Pop()
  {
    if (top == -1)
    {
      cout << "Some error occurred";
    }
    return UI_Elements[top--];
  }

  string Top()
  {
```

```
      return UI_Elements[top];
    }

    int Size()
    {
      return top + 1;
    }

    bool isEmpty()
    {
      return (top == -1) ? true : false;
    }
  };

  int main()
  {
      Stack _stack;

      if (_stack.isEmpty())
      {
        cout << "Stack is empty" << endl;
      }
      // Push elements
      _stack.Push("UI_Element1");
      _stack.Push("UI_Element2");
      // Size of stack
      cout << "Size of stack = " << _stack.Size() << endl;
      // Top element
      cout << _stack.Top() << endl;
      // Pop element
      cout << _stack.Pop() << endl;
      // Top element
      cout << _stack.Top() << endl;


      _getch();
      return 0;
    }
```

How it works...

In this example, we have used the `STACK` data structure to push the various UI elements into the stack. The `STACK` itself is created with the help of an array. While pushing an element, we need to check whether the stack is empty or already has some elements present. While popping elements, we need to delete the element that is at the top of the stack and change the pointer address accordingly. While printing the UI elements of the stack, we traverse the entire stack and display them from the top. Let us consider a game with the following levels: Main Menu, Chapter Select, Level Select, and Play Game. When we want to quit the game, we want the user to select the levels in reverse order. So the first level should be Play Game (Pause State), followed by Level Select, Chapter Select, and finally Main Menu. This can be easily achieved with a stack as explained in the previous example.

Using queues to store data

A queue is an example of a dynamic data structure. This means the size of the queue can be changed at runtime. This is a huge advantage when it comes to programming in games. Queues are enqeued/inserted from the rear of the data structure and dequeued/deleted/pushed out from the front of the data structure. This makes it a **first in first out** (**FIFO**) data structure. Imagine, in a game, we have an inventory but we want to make the player use the first item he has picked up unless he manually changes to a different item. This can be easily achieved by a queue. If we want to design it so that the current item switches to the most powerful item in the inventory, we can use a priority queue for that purpose.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will implement the queue data structure using a linked list. It is very easy to implement a queue and it is a very robust data structure to use:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

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

```
#include <conio.h>

using namespace std;

int main()
{
  queue <string> gunInventory;
  gunInventory.push("AK-47");
  gunInventory.push("BullPup");
  gunInventory.push("Carbine");

  cout << "This is your weapons inventory" << endl << endl;
  cout << "The first gun that you are using is "
    << gunInventory.front() << endl << endl;
  gunInventory.pop();
  cout << "There are currently " << gunInventory.size()
    << " more guns in your inventory. " << endl << endl
    << "The next gun in the inventory is "
    << gunInventory.front() << "." << endl << endl

    << gunInventory.back() << " is the last gun in the inventory."
    << endl;

  _getch();
  return 0;

}
```

How it works...

We have used an STL queue to create the queue structure, or rather use the queue structure. The queue structure, as we know, is important when we need to use the FIFO data structure. As in a First Person Shooter, we may want the user to use the first gun he picks up and the remaining guns be put in the inventory. This is an ideal case for a queue, as explained in the example. The front of the queue structure holds the first gun picked up, or the current gun, and the remaining guns are stored in the inventory in the order in which they were picked up. Sometimes, we do want in our game that if we pick up a gun that is more powerful than the one we are using, it should automatically swap to that. In such a case, we can use a more specialized form of queue called a priority queue, where we need to use the same queue structure but just specify by what parameters the queue is to be sorted.

Using trees to store data

A tree is an example of a non-linear data structure, unlike arrays and linked lists which are linear. A tree is often used in games that require hierarchy. Imagine a car with many parts and all the parts are functional, upgradable, and can be interacted with. In this case, we will create the entire class for the car using a tree data structure. A tree uses a parent-child relationship to traverse between all the nodes.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will be implementing a binary tree. There are many variations of the binary tree. We will be creating the most basic binary tree. It is very easy to add new logic to a binary tree to implement a balanced binary, or AVL tree, and so on:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `CTree.cpp`.
5. Add the following lines of code to it:

```
// Initialize the node with a value and pointers
// to left child
// and right child
struct node
{
  string data_value;
  node *left;
  node *right;
};

class Binary_Tree
{
public:
  Binary_Tree();
  ~Binary_Tree();

  void insert(string key);
  node *search(string key);
```

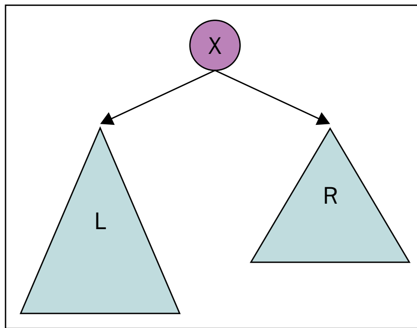
```
    void destroy_tree();

  private:
    void destroy_tree(node *leaf);
    void insert(string key, node *leaf);
    node *search(string key, node *leaf);

    node *root;
  };

  Binary_Tree::Binary_Tree()
  {
    root = NULL;
  }

  Binary_Tree::~Binary_Tree()
  {
    destroy_tree();
  }

  void Binary_Tree::destroy_tree(node *leaf)
  {
    if (leaf != NULL)
    {
      destroy_tree(leaf->left);
      destroy_tree(leaf->right);
      delete leaf;
    }
  }

  void Binary_Tree::insert(string key, node *leaf)
  {
    if (key< leaf->key_value)
    {
      if (leaf->left != NULL)
        insert(key, leaf->left);
      else
      {
        leaf->left = new node;
        leaf->left->key_value = key;
        leaf->left->left = NULL;
        leaf->left->right = NULL;
      }
    }
```

```
  else if (key >= leaf->key_value)
  {
    if (leaf->right != NULL)
      insert(key, leaf->right);
    else
    {
      leaf->right = new node;
      leaf->right->key_value = key;
      leaf->right->left = NULL;
      leaf->right->right = NULL;
    }
  }
}

node *Binary_Tree::search(string key, node *leaf)
{
  if (leaf != NULL)
  {
    if (key == leaf->key_value)
      return leaf;
    if (key<leaf->key_value)
      return search(key, leaf->left);
    else
      return search(key, leaf->right);
  }
  else return NULL;
}

void Binary_Tree::insert(string key)
{
  if (root != NULL)
    insert(key, root);
  else
  {
    root = new node;
    root->key_value = key;
    root->left = NULL;
    root->right = NULL;
  }
}
node *Binary_Tree::search(string key)
{
```

```
        return search(key, root);
    }

    void Binary_Tree::destroy_tree()
    {
        destroy_tree(root);
    }
```

How it works...

We use a struct to store the value and a pointer to the left child and the right child. There is no particular rule as to which elements should be your left child and which elements should be the right child. We can decide, if we want, that all elements lower than the root element should be on the left and all elements greater that the root are on the right.



Both insertion and deletion in the tree data structure are done in a recursive way. To insert elements, we traverse the tree and check if it is empty. If it is empty, we create a new node and add all the corresponding nodes recursively, by checking whether the new node's value is greater than or less than the root node. Searching for an element is done in a similar way. If the element to be searched has a value lower than the root node, we can ignore the entire right-hand section of the tree, as we can see in our `search` function, and keep searching recursively. This reduces the search space considerably and optimizes our algorithm. This means searching for an item at runtime will be faster. Let us say we are creating a game where we need to implement procedural terrain. After the scene is loaded, we can use a binary tree to divide the entire level into sections based on whether they appear on the left or the right. If this information is correctly stored in the tree, then the game camera can use this information to decide which section is rendered and which is not. This also creates a great level of culling optimization. If the parent does not get rendered, we can neglect checking the remainder of the tree for rendering.

Using graphs to store data

In this recipe, we will see how easy it is to store data using the graph data structure. The graph data structure is extremely useful if we have to create a system like Facebook to sell and share our game with friends, and friends of friends. A graph can be implemented in a few ways. The most commonly used method is by using edges and nodes.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will see how we can implement graphs. Graphs are a very good data structure for interconnecting various states and data together with edge conditions. Any social networking algorithm uses the graph data structure in one way or another:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add the `CGraph.h/CGraph.cpp` files.
5. Add the following lines of code to `CGraph.h`:

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

using namespace std;

struct vertex
{
  typedef pair<int, vertex*> ve;
  vector<ve> adj; //cost of edge, destination vertex
  string name;
  vertex(string s)
  {
    name = s;
```

```
    }
};

class graph
{
public:
    typedef map<string, vertex *> vmap;
    vmap work;
    void addvertex(const string&);
    void addedge(const string& from, const string& to, double cost);
};
```

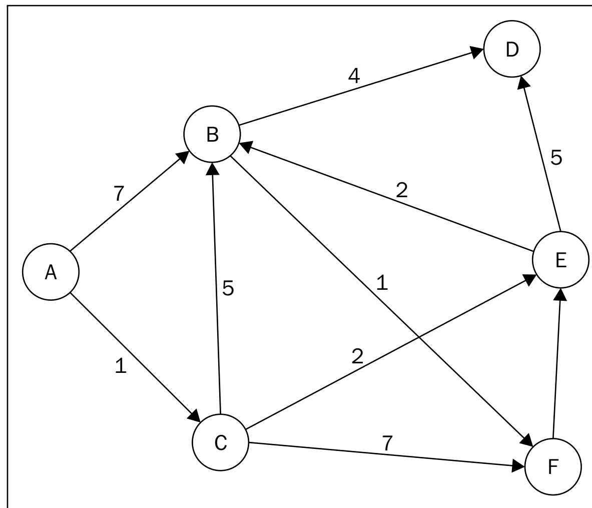
6. Add the following lines of code to `CGraph.cpp`:

```
void graph::addvertex(const string &name)
{
    vmap::iterator itr = work.begin();
    itr = work.find(name);
    if (itr == work.end())
    {
        vertex *v;
        v = new vertex(name);
        work[name] = v;
        return;
    }
    cout << "\nVertex already exists!";
}

void graph::addedge(const string& from, const string& to, double
cost)
{
    vertex *f = (work.find(from)->second);
    vertex *t = (work.find(to)->second);
    pair<int, vertex *> edge = make_pair(cost, t);
    f->adj.push_back(edge);
}
```

How it works...

A graph comprises of edges and nodes. So, the first thing to do while implementing a graph data structure is to create a struct to store the node and vertex information. The following diagram has six nodes and seven edges. To implement a graph, we need to understand the cost of each edge to go from one node to another. These are called adjacency costs. To insert a node, we create a node. To add an edge to the node, we need to supply the information about the two nodes that need to be connected and the cost of the edge.

After we get that information, we create a pair using the cost of the edge and one of the nodes and push that edge information to the other node:



Using STL lists to store data

STL is a standard template library that contains a lot of implementations of the basic data structures, which means we can directly use them for our purpose. The list is internally implemented as a doubly linked list, which means insertion and deletion can happen at both ends.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

www.allitebooks.com

How to do it...

In this recipe, we will see how we can easily use the inbuilt template library provided for us by C++ to create complex data structures. After the complex data structure has been created, we can easily use it to store data and access it:

1. Open Visual Studio.
2. Create a new C++ project.
3. Add a source file called `Source.cpp`.
4. Add the following lines of code to it:

```
#include <iostream>
#include <list>
#include <conio.h>

using namespace std;

int main()
{
  list<int> possible_paths;
  possible_paths.push_back(1);
  possible_paths.push_back(1);
  possible_paths.push_back(8);
  possible_paths.push_back(9);
  possible_paths.push_back(7);
  possible_paths.push_back(8);
  possible_paths.push_back(2);
  possible_paths.push_back(3);
  possible_paths.push_back(3);

  possible_paths.sort();
  possible_paths.unique();

  for (list<int>::iterator list_iter = possible_paths.begin();
    list_iter != possible_paths.end(); list_iter++)
  {
    cout << *list_iter << endl;
  }


  _getch();
  return 0;

}
```

How it works...

We have used the list to push back the values of possible path costs for a certain AI player to reach a destination. We have used an STL list, which comes with a few functions built in that we can apply on the container. We use the `sort` function to sort the list in ascending order. We also have the `unique` function to delete all duplicate values from the list. After sorting the list, we have the least path cost, and accordingly we can apply that path to the AI player. Although the code size is reduced immensely and it is much easier to write, we should use STL with caution as we are never sure about the algorithm behind the inbuilt functions. For example, the `sort` function most likely uses QuickSort but we don't know.

Using STL maps to store data

A map is one of the associative containers of STL and stores elements formed by a combination of a key value and a mapped value, following a specific order. Maps are a part of the STL provided for us by C++.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will see how we can easily use the inbuilt template library provided by C++ to create complex data structures. After the complex data structure is created, we can easily use it to store data and access it:

1. Open Visual Studio.
2. Create a new C++ project.
3. Add a source file called `Source.cpp`.
4. Add the following lines of code to it:

```
#include <iostream>
#include <map>
#include <conio.h>

using namespace std;

int main()
{
  map <string, int> score_list;

  score_list["John"] = 242;
```

```
    score_list["Tim"] = 768;
    score_list["Sam"] = 34;

    if (score_list.find("Samuel") == score_list.end())
    {
      cout << "Samuel is not in the map!" << endl;
    }


    cout << score_list.begin()->second << endl;




    _getch();
    return 0;

  }
```

How it works...

We have used the STL map to create a key/value pair to store the names of the players playing our game, along with their high scores. We can use any data type in a map. In our example, we have used a string and an int. After creating the data structure, it is very easy to find whether a player exists in the database, and we can also sort the map and display the score associated with the player. The second field gives us the values, whereas the first field gives us the key.

Using STL hash tables to store data

The biggest difference between a map and a hash table is that while the map data structure is ordered, a hash table is unordered. Both use the same principle of key/value pairs. The worst case search complexity for an unordered map is *O(N)*, as it is not ordered like a map, which has a complexity of *O(log N)*.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will see how we can easily use the inbuilt template library provided for us by C++ to create complex data structures. After the complex data structure has been created, we can easily use it to store data and access it:

1. Open Visual Studio.
2. Create a new C++ project.
3. Add a source file called `Source.cpp`.
4. Add the following lines of code to it:

```
#include <unordered_map>
#include <string>
#include <iostream>
#include <conio.h>

using namespace std;


int main()
{
  unordered_map<string, string> hashtable;
  hashtable.emplace("Alexander", "23ms");
  hashtable.emplace("Christopher", "21ms");
  hashtable.emplace("Steve", "55ms");
  hashtable.emplace("Amy", "17ms");
  hashtable.emplace("Declan", "999ms");

  cout << "Ping time in milliseconds: " << hashtable["Amy"] <<
endl<<endl;
  cout << "----------------------------------" << endl << endl;

  hashtable.insert(make_pair("Fawad", "67ms"));

  cout << endl<<"Ping time of all player is the server" << endl;
  cout << "------------------------------------" << endl << endl;
  for (auto &itr : hashtable)
  {
    cout << itr.first << ": " << itr.second << endl;
  }

  _getch();
  return 0;
}
```

How it works...

The program calculates the ping time of all players who are currently playing our game on the server. We create a hash table and store all their names and ping times using the `emplace` keyword. We can also insert a new player later with their ping time by using the `make_pair` keyword. After the hash table has been created, we can easily display the ping time of a particular player, or the ping time of all players in the server. We use an iterator to iterate through the hash table. The first parameter gives us the key and the second parameter gives us the value.

4
Algorithms for Game Development

In this chapter, the following recipes will be covered:

- Using sorting techniques to arrange items
- Using searching techniques to look for an item
- Finding the complexity of an algorithm
- Finding the endian-ness of a device
- Using dynamic programming to break down a complex problem
- Using greedy algorithms to solve problems
- Using divide and conquer algorithms to solve problems

Introduction

An algorithm refers to a list of steps that should be applied to perform a task. Searching and sorting algorithms are techniques with which we can search or sort elements in a container. A container, by itself, will have no advantage unless we can search items within that container. Based on certain containers, certain algorithms become more powerful for some containers than others. As an algorithm will run slower on a slower system and faster on a superior system, computation time is not an effective way to measure the effectiveness of an algorithm. Algorithms are rather measured as steps. Games are real-time applications, so the algorithms that will be applied have to be effective for games to be executed at least at 30 frames per second. The ideal frame rate is 60 frames per second.

Using sorting techniques to arrange items

Sorting is a way to arrange items in a container. We can arrange them in ascending or descending order. If we have to implement the high score system and leader board of a game, sorting becomes necessary. In the game, the moment a user achieves a score higher than his previous highest score, we should update that value as the current highest score and push it to a local or online leader board. If it's local, we should arrange all the user's previous high scores in descending order and display the top 10 scores. If it is an online leader board, we need to sort all the users' latest high scores and display the result.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to arrange items in a container using different sorting techniques:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a header file called `Sorting.h`.
5. Add the following lines of code to it:

```
// Bubble Sort
template <class T>
void bubble_sort(T a[], int n)
{
  T temp;
  for (int i = 0; i<n; i++)
  {
    for (int j = 0; j<n - i - 1; j++)
    {
      if (a[j]>a[j + 1])
      {
        temp = a[j];
        a[j] = a[j + 1];
        a[j + 1] = temp;
      }
    }
```

```
  }
}

//Insertion Sort
template <class T>
void insertion_sort(T a[], int n)
{
  T key;
  for (int i = 1; i<n; i++)
  {
    key = a[i];
    int j = i - 1;
    while (j >= 0 && a[j]>key)
    {
      a[j + 1] = a[j];
      j = j - 1;
    }
    a[j + 1] = key;
  }
}

//Selection Sort
template <class T>
int minimum_element(T a, int low, int up)
{
  int min = low;
  while (low<up)
  {
    if (a[low]<a[min])
      min = low;
    low++;
  }
  return min;
}

template <class T>

void selection_sort(T a[], int n)
{
  int i = 0;
  int loc = 0;
  T temp;
  for (i = 0; i<n; i++)
  {
```

```
        loc = minimum_element(a, i, n);
        temp = a[loc];
        a[loc] = a[i];
        a[i] = temp;
      }
    }

    //Quick Sort
    template <class T>
    int partition(T a[], int p, int r)
    {
      T x;
      int i;
      x = a[r];
      i = p - 1;
      for (int j = p; j <= r - 1; j++)
      {
        if (a[j] <= x)
        {
          i = i + 1;
          swap(a[i], a[j]);
        }
      }
      swap(a[i + 1], a[r]);
      return i + 1;
    }
    template <class T>
    void quick_sort(T a[], int p, int r)
    {
      int q;
      if (p<r)
      {
        q = partition(a, p, r);
        quick_sort(a, p, q - 1);
        quick_sort(a, q + 1, r);
      }
    }
```

How it works...

In this example four sorting techniques have been used. The four techniques are **bubble sort**, **selection sort**, **insertion sort**, and **quick sort**.

Bubble sort is a sorting algorithm that works by continuously traversing the container to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The process is continued until no more swaps are required. The average, best, and worst case scenarios have the order of $O(n^2)$.

Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted container is built one entry at a time. It is a very simple algorithm to implement. However, it is not so effective on large sets of data. The worst and average case scenarios have an order of $O(n^2)$ and the best case scenario, when the container is sorted, has an order of $O(n)$.

Selection sort is an algorithm that attempts to place an item in its correct position in the sorted list at every pass. The best, worst, and average case scenarios have an order of $O(n^2)$.

Quick sort is an algorithm that creates a pivot and then sorts the container based on the pivot. Then the pivot is shifted and the process continues. Quick sort is a very effective algorithm and works on almost all real-world data and most modern architectures. It makes excellent use of memory hierarchy. Even the inbuilt standard template library uses a modified version of quick sort for its sorting algorithm. The best and average case scenarios for this algorithm are $O(n*\log n)$ and the worst case is $O(n^2)$.

Using searching techniques to look for an item

Searching techniques are a group of algorithms that involve the process of looking for an item in a container. Searching and sorting go hand in hand. A sorted container will be easier to search. After a container is sorted or ordered, we can apply an appropriate searching algorithm to find an element. Suppose we need to find the names of the guns that have been used to kill more than 25 enemies. If the container stores the values of the name of the gun and total kills associated with that gun, all we need to do is to first sort that container in ascending number of kills made by the guns. Then we can do a linear search in which we find the first gun that has more than 25 kills. Correspondingly, the next items in the container after that will have more than 25 kills, as the container is sorted. However, we can apply better searching techniques.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how we can easily apply searching algorithms to our program:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;

bool Linear_Search(int list[], int size, int key)
{
  // Basic sequential search
  bool found = false;
  int i;

  for (i = 0; i < size; i++)
  {
    if (key == list[i])
      found = true;
    break;
  }

  return found;
}
bool Binary_Search(int *list, int size, int key)
{
  // Binary search
  bool found = false;
  int low = 0, high = size - 1;

  while (high >= low)
  {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key > list[mid])
      low = mid + 1;
```

```
        else
        {
          found = true;
          break;
        }
      }

      return found;
    }
```

How it works...

Searching for items in a container can be done in many ways. However, it matters a lot whether the container has been sorted or not. Let us assume that the container is sorted. The worst way to search for an item is to traverse the whole container and search for the item. This will take a lot of time for large data sets and is absolutely not advisable in games programming. A better way to search for an item is by using binary searching. Binary searching works by dividing the container into two halves. It checks at the midpoint if the value to be searched is less than or greater than the midpoint value. If it is greater, we can ignore the first half of the container and continue searching only in the second half. Again repeat the process for the second half, by further dividing into two halves. Consequently, by doing this we can reduce the search space of the algorithm immensely. The order of this algorithm is O(log n).

Finding the complexity of an algorithm

We need an effective way to measure algorithms. That way we will find out whether our algorithm is effective or not. An algorithm will work more slowly on slower machines and more quickly on faster machines, so computation time is not an effective way to measure algorithms. Algorithms should rather be measured as a number of steps. We call that the order of the algorithm. We also need to find out the best case, worst case, and average case scenarios for the order of the algorithm. This will give us a clearer picture of how our algorithm will be applied to small sets of data and larger sets of data. Complex algorithms or algorithms of a higher order should be avoided, as these will increase the number of steps that the device will need to do to perform the task, and hence will slow down the application. Also, debugging becomes difficult with such algorithms.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how easy it is to find the complexity of an algorithm.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;

void Cubic_Order()
{
  int n = 100;
  for (int i = 0; i < n; i++)
  {
    for (int j=0; j < n; j++)
    {
      for (int k = 0; k < n; k++)
      {
        //Some implementation
      }
    }
  }
}
void Sqaure_Order()
{
  int n = 100;
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
    {
      //Some implementation
    }
  }
}

int main()
{
```

```
    Cubic_Order();
    Sqaure_Order();

    return 0;
}
```

How it works...

In this example, we can see how the order of an algorithm, or the "Big O" notation, varies with implementation. If we take the first function, `Cubic_Order`, the innermost implementation will take *n*n*n* steps to find the answer. So it has an order of n-cubed, *O(n^3)*. This is really bad. Imagine if n is a really large data set, for example let's say *n =1000*, it will take 1,000,000,000 steps to find the solution. Avoid cubic order algorithms whenever you can. The second function, `square_order`, has a square order. The innermost implementation will take *n*n* steps to find a solution, so the order of that algorithm is *O(n^2)*. This is, again, bad practice.

We should attempt to achieve at least *O(log N)* complexity. We can achieve log *N* complexity if we continuously decrease the search space by half, for example, by using binary searching. There are order algorithms that achieve *O (log N)*, which is greatly optimized.

As a general rule, all algorithms following *divide and conquer* will have *O (log N)* complexity.

Finding the endian-ness of a device

The endian-ness of a platform refers to the way the most significant byte is stored on that device. This information is highly important as many algorithms can be optimized based on this information. Notably, the two most popular rendering SDKs, DirectX and OpenGL, differ in their endian-ness. The two different types of endian-ness are called big endian and little endian.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to find the endian-ness of a device.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`.

5. Add the following lines of code to it:

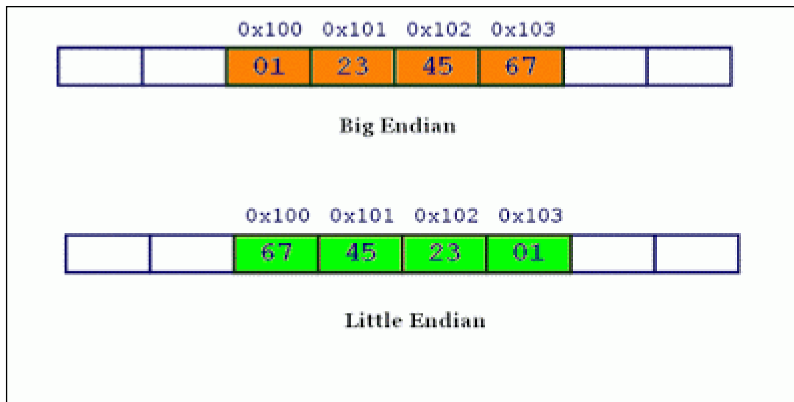
```
Source.cpp

#include <stdio.h>
#include <iostream>
#include <conio.h>

using namespace std;

bool isBigEndian()
{
  unsigned int i = 1;
  char *c = (char*)&i;
  if (*c)
    return false;
  else
    return true;
}
int main()
{
  if (isBigEndian())
  {
    cout << "This is a Big Endian machine" << endl;
  }
  else
  {
    cout << "This is a Little Endian machine" << endl;
  }

  _getch();
  return 0;
}
```

How it works...

Little and big endian are two different ways in which multibyte data types are stored on different machines. In little endian machines, the least significant byte of the multibyte data type is stored first. On the other hand, in big endian machines, the most significant byte of binary representation of the multibyte data type is stored first.

In the preceding program, a character pointer `c` is pointing to an integer `i`. Since the size of character is 1 byte when the character pointer is dereferenced, it will contain only the first byte of the integer. If the machine is little endian, then `*c` will be `1` (because the last byte is stored first), and if the machine is big endian then `*c` will be 0.

Suppose the integer is stored as 4 bytes; then, a variable `x` with value 0x01234567 will be stored as follows:



Most of the time, a compiler takes care of endian-ness; however, endian-ness becomes an issue in network programming if we are sending data from a little endian machine to a big endian machine. Also, it becomes an issue if we switch our rendering pipeline from DirectX to OpenGL.

Using dynamic programming to break down a complex problem

Dynamic programming is a very modern way to solve problems. The process involves breaking a big problem into smaller chunks of problems, finding solutions for those chunks and repeating the process to solve the entire complex problem. It is a bit difficult to grasp this technique at first, but with sufficient practice any problem can be solved using dynamic programming. Most of the problems we will encounter while programming a video game will be complex. Hence, mastering this technique will be really useful.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to use dynamic programming to solve a problem:

1. Open Visual Studio.
2. Create a new C++ project.

3. Select **Win32 Console Application**.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include<iostream>
#include <conio.h>

using namespace std;


int max(int a, int b) { return (a > b) ? a : b; }


int knapSack(int TotalWeight, int individual_weight[], int
individual_value[], int size)
{

  if (size == 0 || TotalWeight == 0)
    return 0;
  if (individual_weight[size - 1] > TotalWeight)
    return knapSack(TotalWeight, individual_weight, individual_
value, size - 1);
  else return max(individual_value[size - 1] +
knapSack(TotalWeight - individual_weight[size - 1], individual_
weight, individual_value, size - 1),
    knapSack(TotalWeight, individual_weight, individual_value,
size - 1)
    );
}


int main()
{
  int individual_value[] = { 60, 100, 120 };
  int individual_weight[] = { 10, 25, 40 };
  int  TotalWeight = 60;
  int size = sizeof(individual_value) / sizeof(individual_
weight[0]);
  cout << "Total value of sack "<<knapSack(TotalWeight,
individual_weight, individual_value, size);

  _getch();
  return 0;
}
```

How it works...

This is an example of the classical *knapsack* problem. This can be applied to many scenarios in game programming, especially for AI resource management. Let us consider that the total weight (sack) that the AI can carry is a constant. In our example, this is the total weight of the knapsack. Every item that the AI collects in the game has a weight and a value. The AI now needs to decide how to fill up his inventory/sack so that he can sell the total sack for maximum value and get coins.

We solve the problem by recursion by solving for every small combination of items (weight and value) and checking for the maximum value of the two combinations, and repeating the process until the total weight of the knapsack is reached.

Using greedy algorithms to solve problems

A greedy algorithm works by finding the optimal solution at every stage. So before processing the next step, it will decide its next step based on the previous outcome and the current needs of the application. In this way, it is better than dynamic programming. However, we cannot apply this principle to all problems, hence a greedy algorithm cannot be used for all situations.

Getting ready

To go through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to use greedy algorithm to solve a problem:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select **Win32 Console Application**.
4. Add the `Source.cpp` files.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;
```

```
void printMaxActivities(int start_Time[], int finish_Time[], int
n)
{
  int i, j;
  i = 0;
  cout << i;
  for (j = 1; j < n; j++)
  {
    if (start_Time[j] >= finish_Time[i])
    {
      cout << j;
      i = j;
    }
  }
}


int main()
{
  int start_Time[] = { 0, 2, 4, 7, 8, 11 };
  int finish_Time[] = { 2, 4, 6, 8, 9, 15 };
  int n = sizeof(start_Time) / sizeof(start_Time[0]);
  printMaxActivities(start_Time, finish_Time, n);

  _getch();
  return 0;
}
```

How it works...

In this example, we have a set of start times and finish times for different activities. We need to find out which activities can be performed by a single person. We can assume that the container is already sorted based on the finish time. So at every pass, we check whether the current start time is greater than or equal to the previous finish time. Only then can we take up the task. We traverse the container and keep checking the same condition. Because we are checking at every step, this algorithm is pretty well optimized.

Using divide and conquer algorithms to solve problems

In general, divide and conquer is based on the following idea. The whole problem we want to solve may be too big to understand or solve at once. We break it up into smaller pieces, solve the pieces separately, and combine the separate pieces.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to use a greedy algorithm to solve a problem:

1. Open Visual Studio.
2. Create a new C++ project.
3. Add a source file called `Source.cpp`.
4. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;

const int MAX = 10;

class rray
{
private:
  int arr[MAX];
  int count;
public:
  array();
  void add(int num);
  void makeheap(int);
  void heapsort();
  void display();
};
array ::array()
{
  count = 0;
  for (int i = 0; i < MAX; i++)
    arr[MAX] = 0;
}
void array ::add(int num)
{
  if (count < MAX)
  {
    arr[count] = num;
```

```
      count++;
    }
    else
      cout << "\nArray is full" << endl;
  }
  void array ::makeheap(int c)
  {

    for (int i = 1; i < c; i++)
    {
      int val = arr[i];
      int s = i;
      int f = (s - 1) / 2;
      while (s > 0 && arr[f] < val)
      {
        arr[s] = arr[f];
        s = f;
        f = (s - 1) / 2;
      }
      arr[s] = val;
    }
  }
  void array ::heapsort()
  {
    for (int i = count - 1; i > 0; i--)
    {
      int ivalue = arr[i];
      arr[i] = arr[0];
      arr[0] = ivalue;
      makeheap(i);

    }
  }
  void array ::display()
  {
    for (int i = 0; i < count; i++)
      cout << arr[i] << "\t";
    cout << endl;
  }
  void main()
  {
    array a;

    a.add(11);
```

```
    a.add(2);
    a.add(9);
    a.add(13);
    a.add(57);
    a.add(25);
    a.add(17);
    a.add(1);
    a.add(90);
    a.add(3);
    a.makeheap(10);
    cout << "\nHeap Sort.\n";
    cout << "\nBefore Sorting:\n";
    a.display();
    a.heapsort();
    cout << "\nAfter Sorting:\n";
    a.display();

    _getch();
  }
```

How it works...

A **heap sorting algorithm** works by first organizing the data to be sorted into a special type of binary tree called a **heap**. The heap itself has, by definition, the largest value at the top of the tree, so the heap sorting algorithm must also reverse the order. It does this with the following steps:

1. Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item is stored in an array.
2. Re-establish the heap.
3. Repeat steps 1 and 2 until there are no more items left in the heap. The sorted elements are now stored in an array.

5

Event-Driven Programming – Making Your First 2D Game

In this chapter, the following recipes will be covered:

- Starting to make a Windows game
- Using Windows classes and handles
- Creating your first window
- Adding keyboard and mouse controls with text output
- Using Windows resources with GDI
- Using dialogs and controls
- Using sprites
- Using animated sprites

Introduction

Windows programming is the start of creating proper applications. We need to know how to package our game into one executable file so that all our resources, such as images, models, and sounds, are encrypted properly and packaged into one file. By doing this, we make sure that the files are safe and cannot be illegally copied on distribution. The application, however, still makes use of these files at runtime.

Windows programming also marks the start of understanding the Windows Message Pump. This system is very important to understand, as all major programming paradigms depend on this principle, especially when we are doing event-driven programming.

The main principle of event driven programming is that, based on events, we should process something. The concept to be understood here is how often we check for events and how often we should process them.

Starting to make a Windows game

The first thing to understand before we start making a Windows game is how a window or a message box is drawn. We need to be aware of the numerous inbuilt functions that Windows provides us with and the different callback functions that we can use.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. There are no other prerequisites.

How to do it...

In this recipe, we will see how easy it is to create a message box in Windows. There are different types of message box we can create, and it is only a matter of a few lines of code. Follow these steps:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to `Source.cpp`:

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <windowsx.h>

int WINAPI WinMain(HINSTANCE _hInstance,
  HINSTANCE _hPrevInstance,
  LPSTR _lpCmdLine,
  int _iCmdShow)
{
```

```
    MessageBox(NULL, L"My first message",
      L"My first Windows Program",
      MB_OK | MB_ICONEXCLAMATION);

    return (0);
  }
```

How it works...

`WINMAIN()` is the entry point of a Windows program. In this example, we have used the inbuilt function to create a message box. `windows.h` contains all the necessary files that we need to call the inbuilt functions present in the Windows API. A message box is typically used to display something. We can also associate message boxes with default Windows sounds. The display of the message box can also be controlled to a great extent. We need to use the right type of parameter to achieve this.

There are other types of message box that we can use as well:

- **MB_OK**: One button, with the **OK** message
- **MB_OKCANCEL**: Two buttons, with the **OK**, **Cancel** message
- **MB_RETRYCANCEL**: Two buttons, with the **Retry**, **Cancel** message
- **MB_YESNO**: Two buttons, with the **Yes**, **No** message
- **MB_YESNOCANCEL**: Three buttons, with the **Yes**, **No**, **Cancel** message
- **MB_ABORTRETRYIGNORE**: Three buttons, with the **Abort**, **Retry**, **Ignore** message
- **MB_ICONEXCLAIMATION**: An exclamation point icon appears
- **MB_ICONINFORMATION**: An information icon appears
- **MB_ICONQUESTION**: A question mark icon appears
- **MB_ICONSTOP**: A stop sign icon appears

Like all good Win32 or Win64 API functions, `MessageBox` returns a value to let us know what happened.

Using Windows classes and handles

To write games, we do not need to know a lot about Windows programming. What we need to know is how to open a window, how to process messages, and how to call the main game loop. The first task of a Windows application is to create a window. After the windows is created, we can do various other things, such as processing events and handling callbacks. These events are finally used by the game framework to display sprites on the screen and make them movable and interactive so that we can play a game.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how easy it is to use Windows classes and handles.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
   // This only adds the necessary windows files and not all of
them
#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // Include all the windows headers.
#include <windowsx.h>  // Include useful macros.

#define WINDOW_CLASS_NAME L"WINCLASS1"

void GameLoop()
{
  //One frame of game logic occurs here...
}

LRESULT CALLBACK WindowProc(HWND _hwnd,
  UINT _msg,
  WPARAM _wparam,
  LPARAM _lparam)
{
  // This is the main message handler of the system.
  PAINTSTRUCT ps; // Used in WM_PAINT.
  HDC hdc;        // Handle to a device context.

  // What is the message?
  switch (_msg)
  {
```

```
  case WM_CREATE:
  {
            // Do initialization stuff here.

            // Return Success.
            return (0);
  }
    break;

  case WM_PAINT:
  {
           // Simply validate the window.
           hdc = BeginPaint(_hwnd, &ps);

           // You would do all your painting here...

           EndPaint(_hwnd, &ps);

           // Return Success.
           return (0);
  }
    break;

  case WM_DESTROY:
  {
             // Kill the application, this sends a WM_QUIT
message.
             PostQuitMessage(0);

             // Return success.
             return (0);
  }
    break;

  default:break;
  } // End switch.

  // Process any messages that we did not take care of...

  return (DefWindowProc(_hwnd, _msg, _wparam, _lparam));
}

int WINAPI WinMain(HINSTANCE _hInstance,
  HINSTANCE _hPrevInstance,
```

```
    LPSTR _lpCmdLine,
    int _nCmdShow)
  {
    WNDCLASSEX winclass; // This will hold the class we create.
    HWND hwnd;           // Generic window handle.
    MSG msg;             // Generic message.

    // First fill in the window class structure.
    winclass.cbSize = sizeof(WNDCLASSEX);
    winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_
  VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra = 0;
    winclass.cbWndExtra = 0;
    winclass.hInstance = _hInstance;
    winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground =
      static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;
    winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register the window class
    if (!RegisterClassEx(&winclass))
    {
      return (0);
    }

    // create the window
    hwnd = CreateWindowEx(NULL, // Extended style.
      WINDOW_CLASS_NAME,      // Class.
      L"My first Window",   // Title.
      WS_OVERLAPPEDWINDOW | WS_VISIBLE,
      0, 0,                    // Initial x,y.
      400, 400,                // Initial width, height.
      NULL,                   // Handle to parent.
      NULL,                   // Handle to menu.
      _hInstance,             // Instance of this application.
      NULL);                  // Extra creation parameters.

    if (!(hwnd))
    {
      return (0);
```

```
    }

    // Enter main event loop
    while (true)
    {
      // Test if there is a message in queue, if so get it.
      if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
        // Test if this is a quit.
        if (msg.message == WM_QUIT)
        {
          break;
        }

        // Translate any accelerator keys.
        TranslateMessage(&msg);
        // Send the message to the window proc.
        DispatchMessage(&msg);
      }

      // Main game processing goes here.
      GameLoop(); //One frame of game logic occurs here...
    }

    // Return to Windows like this...
    return (static_cast<int>(msg.wParam));
  }
```

How it works...

The entire `typedef` struct `_WNDCLASSEX` is defined as follows:

```
{
UINT cbSize;          // Size of this structure.
UINT style;           // Style flags.
WNDPROC lpfnWndProc;  // Function pointer to handler.
int cbClsExtra;       // Extra class info.
int cbWndExtra;       // Extra window info.
HANDLE hInstance;     // The instance of the app.
HICON hIcon;          // The main icon.
HCURSOR hCursor;      // The cursor for the window.
HBRUSH hbrBackground; // The Background brush to paint the window.
```

```
LPCTSTR lpszMenuName; // The name of the menu to attach.
LPCTSTR lpszClassName;// The name of the class itself.
HICON hIconSm;        // The handle of the small icon.
} WNDCLASSEX;
```

The Windows API provides us with multiple API callbacks. We need to decide which message to intercept and what information to process in that message pump. For example, `WM_CREATE` is a Windows create function. We should perform most of our initializations here. Similarly, `WM_DESTROY` is where we need to destroy our created objects. We need to use GDI objects to paint boxes and other things on the window. We can also display our own cursors and icons on the window.

Creating your first window

Creating a window is the first step in Windows programming. All our sprites and other objects will be drawn on top of this window. There is a standard way of drawing a window. So this part of the code will be repeated in all programs that use Windows programming to draw something.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how easy it is to create a window:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // Include all the windows headers.
#include <windowsx.h>  // Include useful macros.
#include "resource.h"

#define WINDOW_CLASS_NAME L"WINCLASS1"


void GameLoop()
```

```
{
  //One frame of game logic occurs here...
}

LRESULT CALLBACK WindowProc(HWND _hwnd,
  UINT _msg,
  WPARAM _wparam,
  LPARAM _lparam)
{
  // This is the main message handler of the system.
  PAINTSTRUCT ps; // Used in WM_PAINT.
  HDC hdc;        // Handle to a device context.

  // What is the message?
  switch (_msg)
  {
  case WM_CREATE:
  {
            // Do initialization stuff here.

            // Return Success.
            return (0);
  }
    break;

  case WM_PAINT:
  {
           // Simply validate the window.
           hdc = BeginPaint(_hwnd, &ps);

           // You would do all your painting here...

           EndPaint(_hwnd, &ps);

           // Return Success.
           return (0);
  }
    break;

  case WM_DESTROY:
  {
             // Kill the application, this sends a WM_QUIT
message.
```

```
                PostQuitMessage(0);

                // Return success.
                return (0);
      }
        break;


      default:break;
      } // End switch.

      // Process any messages that we did not take care of...

      return (DefWindowProc(_hwnd, _msg, _wparam, _lparam));
    }

    int WINAPI WinMain(HINSTANCE _hInstance,
      HINSTANCE _hPrevInstance,
      LPSTR _lpCmdLine,
      int _nCmdShow)
    {
      WNDCLASSEX winclass; // This will hold the class we create.
      HWND hwnd;           // Generic window handle.
      MSG msg;             // Generic message.

      HCURSOR hCrosshair = LoadCursor(_hInstance, MAKEINTRESOURCE(IDC_
    CURSOR2));

      // First fill in the window class structure.
      winclass.cbSize = sizeof(WNDCLASSEX);
      winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_
    VREDRAW;
      winclass.lpfnWndProc = WindowProc;
      winclass.cbClsExtra = 0;
      winclass.cbWndExtra = 0;
      winclass.hInstance = _hInstance;
      winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      winclass.hCursor = LoadCursor(_hInstance, MAKEINTRESOURCE(IDC_
    CURSOR2));
      winclass.hbrBackground =
        static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
      winclass.lpszMenuName = NULL;
      winclass.lpszClassName = WINDOW_CLASS_NAME;
```

```
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

// register the window class
if (!RegisterClassEx(&winclass))
{
  return (0);
}

// create the window
hwnd = CreateWindowEx(NULL, // Extended style.
  WINDOW_CLASS_NAME,      // Class.
  L"Packt Publishing",    // Title.
  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
  0, 0,                    // Initial x,y.
  400, 400,                // Initial width, height.
  NULL,                   // Handle to parent.
  NULL,                   // Handle to menu.
  _hInstance,             // Instance of this application.
  NULL);                  // Extra creation parameters.

if (!(hwnd))
{
  return (0);
}

// Enter main event loop
while (true)
{
  // Test if there is a message in queue, if so get it.
  if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
  {
    // Test if this is a quit.
    if (msg.message == WM_QUIT)
    {
      break;
    }

    // Translate any accelerator keys.
    TranslateMessage(&msg);
    // Send the message to the window proc.
    DispatchMessage(&msg);
  }

  // Main game processing goes here.
```

```
        GameLoop(); //One frame of game logic occurs here...
    }

    // Return to Windows like this...
    return (static_cast<int>(msg.wParam));
}
```

How it works...

In this example, we have used the standard Windows API callback. We query on the message parameter that is passed and, based on that, we intercept and perform suitable actions. We have used the `WM_PAINT` message to paint the window for us and the `WM_DESTROY` message to destroy the current window. To paint the window, we need a handle to the device context and then we can use `BeginPaint` and `EndPaint` appropriately. In the main structure, we need to fill up the Windows structures and specify the current cursor and icons that need to be loaded. Here, we can specify what color brush we are going to use to paint the window. Finally, the size of the window is specified and registered. After that, we need to continuously peek messages, translate them, and finally dispatch them to the Windows procedure.

Adding keyboard and mouse controls with text output

One of the most important things that we require in a video game is a human interface to interact with. The most common interface devices are the keyboard and the mouse. Therefore, it is very important to understand how they work and how we can detect key presses and movements. It is equally important to know how to display specific text on the screen; this can be really useful for debugging and for HUD implementation.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to detect keyboard and mouse events:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.
4. Add a source file called `Source.cpp`.

5. Add the following lines of code to it:

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h> //Include all the Windows headers.
#include <windowsx.h> //Include useful macros.
#include <strstream>
#include <string>
#include <cmath>

#include "resource.h"
#include "mmsystem.h"
//also uses winmm.lib

using namespace std;

#define WINDOW_CLASS_NAME "WINCLASS1"

HINSTANCE g_hInstance;
//RECT g_rect;
const RECT* g_prect;

POINT g_pos;
int g_iMouseX;
int g_iMouseY;

bool IS_LEFT_PRESSED  = 0;
bool IS_RIGHT_PRESSED = 0;
bool IS_UP_PRESSED    = 0;
bool IS_DOWN_PRESSED  = 0;

bool IS_LMB_PRESSED = 0;
bool IS_RMB_PRESSED = 0;
bool IS_MMB_PRESSED = 0;

int LAST_KEYPRESS_ASCII = 0;

float ang = 0.0f;

template<typename T>
std::string ToString(const T& _value)
{
  std::strstream theStream;
```

```
  theStream << _value << std::ends;
  return (theStream.str());
}



//GameLoop
void GameLoop()
{
  ang += 0.0005f;
  //One frame of game logic goes here
}

//Event handling (window handle, message handle --
LRESULT CALLBACK WindowProc(HWND _hwnd, UINT _msg, WPARAM _wparam,
LPARAM _lparam)
{
  //This is the main message handler of the system.
  PAINTSTRUCT ps; //Used in WM_PAINT
  HDC hdc;        // Handle to a device context.

      if ((GetAsyncKeyState(VK_LEFT) & 0x8000) == 0x8000)
      {
        IS_LEFT_PRESSED = TRUE;
      }
      else
      {
        IS_LEFT_PRESSED = FALSE;
      }

      if ((GetAsyncKeyState(VK_RIGHT) & 0x8000) == 0x8000)
      {
        IS_RIGHT_PRESSED = TRUE;
      }
      else
      {
        IS_RIGHT_PRESSED = FALSE;
      }

      if ((GetAsyncKeyState(VK_UP) & 0x8000) == 0x8000)
      {
        IS_UP_PRESSED = TRUE;
      }
      else
```

```
        {
          IS_UP_PRESSED = FALSE;
        }

        if ((GetAsyncKeyState(VK_DOWN) & 0x8000) == 0x8000)
        {
          IS_DOWN_PRESSED = TRUE;
        }
        else
        {
          IS_DOWN_PRESSED = FALSE;
        }

    //What is the message?
    switch(_msg)
    {
    case WM_CREATE:
      {
        //Do initialisation stuff here.
        //Return success.
        return(0);
      }
      break;

    case WM_PAINT:
      {
        ////Simply validate the window.
        hdc = BeginPaint(_hwnd, &ps);

        InvalidateRect( _hwnd,
          g_prect,
          FALSE);


        string temp;
        int iYDrawPos = 15;

        COLORREF red = RGB(255,0,0);

        SetTextColor(hdc, red);

        temp = "MOUSE X: ";
        temp += ToString((g_pos.x));
        while (temp.size() < 14)
```

```
        {
          temp += " ";
        }

        TextOut(hdc,30,iYDrawPos,temp.c_str(), static_
  cast<int>(temp.size()));

        iYDrawPos+= 13;

        temp = "MOUSE Y: ";
        temp += ToString((g_pos.y));
        while (temp.size() < 14)
        {
          temp += " ";
        }

        TextOut(hdc,30,iYDrawPos,temp.c_str(), static_
  cast<int>(temp.size()));

        iYDrawPos+= 13;

        if (IS_LEFT_PRESSED == TRUE)
        {
          TextOut(hdc,30,iYDrawPos,"LEFT IS PRESSED", 24);
        }
        else
        {
          TextOut(hdc,30,iYDrawPos,"LEFT IS NOT PRESSED ", 20);
        }
        iYDrawPos+= 13;
        if (IS_RIGHT_PRESSED == TRUE)
        {
          TextOut(hdc,30,iYDrawPos,"RIGHT IS PRESSED", 25);
        }
        else
        {
          TextOut(hdc,30,iYDrawPos,"RIGHT IS NOT PRESSED ", 21);
        }
        iYDrawPos+= 13;
        if (IS_DOWN_PRESSED == TRUE)
        {
          TextOut(hdc,30,iYDrawPos,"DOWN IS PRESSED", 24);
        }
        else
```

```
        {
          TextOut(hdc,30,iYDrawPos,"DOWN IS NOT PRESSED", 20);
        }
        iYDrawPos+= 13;
        if (IS_UP_PRESSED == TRUE)
        {
          TextOut(hdc,30,iYDrawPos,"UP IS PRESSED", 22);
        }
        else
        {
          TextOut(hdc,30,iYDrawPos,"UP IS NOT PRESSED ", 18);
        }


//      TextOut(hdc, static_cast<int>(200 +(sin(ang)*200)),
static_cast<int>(200 +(sin(ang)*200))) , "O", 1);

        EndPaint(_hwnd, &ps);

        //Return success.
        return(0);
      }
      break;

    case WM_DESTROY:
      {
        //Kill the application, this sends a WM_QUIT message.
        PostQuitMessage(0);

        //Return Sucess.
        return(0);
      }
      break;

    case WM_MOUSEMOVE:
      {
        GetCursorPos(&g_pos);
        // here is your coordinates
        //int x=pos.x;
        //int y=pos.y;
        return(0);
      }
```

```
    break;


    case WM_COMMAND:
      {

      }

    default:break;
    } // End switch.

    //Process any messages we didn't take care of...

    return(DefWindowProc(_hwnd, _msg, _wparam, _lparam));
  }

int WINAPI WinMain(HINSTANCE _hInstance, HINSTANCE _hPrevInstance,
LPSTR _lpCmdLine, int _nCmdShow)
{
  WNDCLASSEX winclass; ///This will hold the class we create
  HWND hwnd; //Generic window handle.
  MSG msg; //Generic message.

  g_hInstance = _hInstance;

  //First fill in the window class structure
  winclass.cbSize         = sizeof(WNDCLASSEX);
  winclass.style          = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW |
CS_VREDRAW;
  winclass.lpfnWndProc    = WindowProc;
  winclass.cbClsExtra     = 0;
  winclass.cbWndExtra     = 0;
  winclass.hInstance      = _hInstance;
  winclass.hIcon          = LoadIcon(g_hInstance,
MAKEINTRESOURCE(IDI_ICON1));
  winclass.hCursor        = NULL;
  winclass.hbrBackground  = static_cast<HBRUSH>(GetStockObject(WHI
TE_BRUSH));
  winclass.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU1);
  winclass.lpszClassName  = WINDOW_CLASS_NAME;
  winclass.hIconSm        = LoadIcon(g_hInstance,
MAKEINTRESOURCE(IDI_ICON1));

  //Register the window class
```

```
if (!RegisterClassEx(&winclass))
{ //perhaps use log manager here
  return(0);
}

//Create the window
if (!(hwnd = CreateWindowEx(NULL, //Extended style.
              WINDOW_CLASS_NAME, //Class
              "Recipe4", //Title
              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
              400,300, //Initial X, Y
              400,400, //Initial width, height.
              NULL, //handle to parent.
              NULL, //handle to menu
              _hInstance, //Instance of this application
              NULL))) //Extra creation parameters
{
  return (0);
}

RECT rect;
rect.left = 0;
rect.right = 400;
rect.top = 0;
rect.bottom = 400;
g_prect = &rect;

//Enter main event loop
while (TRUE)
{
  //Test if there is a message in queue, if so get it.
  if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
  {
    //Test if this is a quit
    if (msg.message == WM_QUIT)
    {
      break;
    }

    //Translate any accelerator keys
    TranslateMessage(&msg);
    //Send the message to the window proc.
```

```
            DispatchMessage(&msg);
          }

          //Main game processing goes here.
          GameLoop(); //One frame of game logic goes here...
        }
        //Return to Windows like this...
        return(static_cast<int>(msg.wParam));
      }
```

How it works...

The main window is created and registered. In the callback function, we use a function called `GetAsyncKeyState(VK_KEYNAME)` to detect which key was pressed. After that, we perform a bitwise `AND` operation to check whether the last key press was also the same key and whether it has actually been pressed. We then have different Boolean parameters to detect the state of the key press and store them. The code could be structured in a better way, but this is the easiest way to understand how to detect key presses. To detect the mouse movement coordinates, we use a function called `GetCursorPos` inside `WM_MOUSEMOVE` and accordingly get the *x* and *y* coordinates on screen. Finally, we need to display all this information on the screen. To do this, we create a rectangle on screen. In that rectangle, we need to use a function called `TextOut` to display that information. The `TextOut` function uses the handle to the device context, the *x* and *y* coordinates, and the message to be displayed.

Using Windows resources with GDI

Graphics Device Interface (**GDI**) allows us to do interesting things using bitmaps, icons cursors, and so on. GDI is used as a rendering alternative if we are not implementing any other rendering alternatives such as OpenGL or DirectX.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to load resources using the Windows GDI:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.

4. Right-click on **Resource files** and add a new cursor from the **Add Resource** subsection.
5. A `resource.h` file will automatically be created for you.
6. Add a source file called `Source.cpp` and add the following code to it:

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // Include all the windows headers.
#include <windowsx.h>  // Include useful macros.
#include "resource.h"

#define WINDOW_CLASS_NAME L"WINCLASS1"


void GameLoop()
{
  //One frame of game logic occurs here...
}

LRESULT CALLBACK WindowProc(HWND _hwnd,
  UINT _msg,
  WPARAM _wparam,
  LPARAM _lparam)
{
  // This is the main message handler of the system.
  PAINTSTRUCT ps; // Used in WM_PAINT.
  HDC hdc;        // Handle to a device context.

  // What is the message?
  switch (_msg)
  {
  case WM_CREATE:
  {
          // Do initialization stuff here.

          // Return Success.
          return (0);
  }
    break;

  case WM_PAINT:
  {
         // Simply validate the window.
```

```
            hdc = BeginPaint(_hwnd, &ps);

            // You would do all your painting here...

            EndPaint(_hwnd, &ps);

            // Return Success.
            return (0);
  }
    break;

  case WM_DESTROY:
  {
             // Kill the application, this sends a WM_QUIT
message.
             PostQuitMessage(0);

             // Return success.
             return (0);
  }
    break;


  default:break;
  } // End switch.

  // Process any messages that we did not take care of...

  return (DefWindowProc(_hwnd, _msg, _wparam, _lparam));
}

int WINAPI WinMain(HINSTANCE _hInstance,
  HINSTANCE _hPrevInstance,
  LPSTR _lpCmdLine,
  int _nCmdShow)
{
  WNDCLASSEX winclass; // This will hold the class we create.
  HWND hwnd;           // Generic window handle.
  MSG msg;             // Generic message.

  HCURSOR hCrosshair = LoadCursor(_hInstance, MAKEINTRESOURCE(IDC_
CURSOR2));

  // First fill in the window class structure.
```

```
  winclass.cbSize = sizeof(WNDCLASSEX);
  winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_
VREDRAW;
  winclass.lpfnWndProc = WindowProc;
  winclass.cbClsExtra = 0;
  winclass.cbWndExtra = 0;
  winclass.hInstance = _hInstance;
  winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  winclass.hCursor = LoadCursor(_hInstance, MAKEINTRESOURCE(IDC_
CURSOR2));
  winclass.hbrBackground =
    static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
  winclass.lpszMenuName = NULL;
  winclass.lpszClassName = WINDOW_CLASS_NAME;
  winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

  // register the window class
  if (!RegisterClassEx(&winclass))
  {
    return (0);
  }

  // create the window
  hwnd = CreateWindowEx(NULL, // Extended style.
    WINDOW_CLASS_NAME,      // Class.
    L"PacktUp Publishing",   // Title.
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0, 0,                    // Initial x,y.
    400, 400,                // Initial width, height.
    NULL,                   // Handle to parent.
    NULL,                   // Handle to menu.
    _hInstance,             // Instance of this application.
    NULL);                  // Extra creation parameters.

  if (!(hwnd))
  {
    return (0);
  }

  // Enter main event loop
  while (true)
  {
    // Test if there is a message in queue, if so get it.
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
```

```
        {
          // Test if this is a quit.
          if (msg.message == WM_QUIT)
          {
            break;
          }

          // Translate any accelerator keys.
          TranslateMessage(&msg);
          // Send the message to the window proc.
          DispatchMessage(&msg);
        }

        // Main game processing goes here.
        GameLoop(); //One frame of game logic occurs here...
      }

      // Return to Windows like this...
      return (static_cast<int>(msg.wParam));
    }
```

How it works...

Loading up the new cursor is the easiest task to achieve. We need to modify the following line:

```
winclass.hCursor = LoadCursor(_hInstance, MAKEINTRESOURCE(IDC_
CURSOR2))
```

If we specify null here, the default windows cursor will be loaded. Instead, we can load the cursor that we just created. Make sure the reference name of the cursor in `resource.h` is specified as `IDC_CURSOR2`. We can name it anything we want, but we need to call the appropriate reference from the `LoadCursor` function. `MAKEINTRESOURCE` enables us to relate to the resource file from the source code. Similarly, we can load multiple cursors and switch them at runtime if we so desire. The same process is used for loading other resources, such as icons and other bitmaps. When we modify a resource file, the corresponding `resource.h` file must be closed or it will not allow us to edit it. Similarly, if we want to manually edit the `source.h` file, we need to close the corresponding `.rc` or resource file.

Using dialogs and controls

Dialogs are one of the mandatory features of Windows programming. If we are creating a complete application, there will be a stage when we will require dialogs in some form. Dialogs could be in the form of edit boxes, radio buttons, check boxes, and so on. Dialogs come in two forms: modal and modeless. Modal dialog boxes require an immediate response, whereas modeless dialog boxes are more like floating boxes and do not require an immediate response.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to create dialog boxes.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 windows application.
4. Create a new resource file.
5. Select dialog as the type of resource.
6. Edit the box in whatever way you desire.
7. A corresponding `resource.h` file will be created.
8. Add the following code to `Source.cpp` file:

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // Include all the windows headers.
#include <windowsx.h>  // Include useful macros.
#include "resource.h"
#define WINDOW_CLASS_NAME L"WINCLASS1"

void GameLoop()
{
  //One frame of game logic occurs here...
}

BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT msg, WPARAM wparam,
LPARAM lparam)
```

```
{
  switch (msg)
  {
    case WM_INITDIALOG:
      break;
    case WM_COMMAND:
      switch (LOWORD(wparam))
      {
      case IDOK:
        EndDialog(
          hDlg, //Handle to the dialog to end.
          0);   //Return code.
        break;
      case IDCANCEL:
        EndDialog(
          hDlg, //Handle to the dialog to end.
          0);   //Return code.
        break;
      default:
        break;
      }

  }

  return true;
}

LRESULT CALLBACK WindowProc(HWND _hwnd,
  UINT _msg,
  WPARAM _wparam,
  LPARAM _lparam)
{
  // This is the main message handler of the system.
  PAINTSTRUCT ps; // Used in WM_PAINT.
  HDC hdc;        // Handle to a device context.

  // What is the message?
  switch (_msg)
  {
  case WM_CREATE:
  {
            // Do initialization stuff here.

            // Return Success.
```

```
            return (0);
    }
      break;

    case WM_PAINT:
    {
            // Simply validate the window.
            hdc = BeginPaint(_hwnd, &ps);

            // You would do all your painting here...

            EndPaint(_hwnd, &ps);

            // Return Success.
            return (0);
    }
      break;

    case WM_DESTROY:
    {
              // Kill the application, this sends a WM_QUIT
message.
              PostQuitMessage(0);

              // Return success.
              return (0);
    }
      break;

    default:break;
    } // End switch.

    // Process any messages that we did not take care of...

    return (DefWindowProc(_hwnd, _msg, _wparam, _lparam));
}

int WINAPI WinMain(HINSTANCE _hInstance,
    HINSTANCE _hPrevInstance,
    LPSTR _lpCmdLine,
    int _nCmdShow)
{
    WNDCLASSEX winclass; // This will hold the class we create.
    HWND hwnd;           // Generic window handle.
```

```
  MSG msg;             // Generic message.

  // First fill in the window class structure.
  winclass.cbSize = sizeof(WNDCLASSEX);
  winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_
VREDRAW;
  winclass.lpfnWndProc = WindowProc;
  winclass.cbClsExtra = 0;
  winclass.cbWndExtra = 0;
  winclass.hInstance = _hInstance;
  winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
  winclass.hbrBackground =
    static_cast<HBRUSH>(GetStockObject(BLACK_BRUSH));
  winclass.lpszMenuName = NULL;
  winclass.lpszClassName = WINDOW_CLASS_NAME;
  winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

  // register the window class
  if (!RegisterClassEx(&winclass))
  {
    return (0);
  }

  // create the window
  hwnd = CreateWindowEx(NULL, // Extended style.
    WINDOW_CLASS_NAME,      // Class.
    L"My first Window",   // Title.
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0, 0,                   // Initial x,y.
    1024, 980,               // Initial width, height.
    NULL,                   // Handle to parent.
    NULL,                   // Handle to menu.
    _hInstance,             // Instance of this application.
    NULL);                  // Extra creation parameters.

  if (!(hwnd))
  {
    return (0);
  }

  DialogBox(_hInstance, MAKEINTRESOURCE(IDD_DIALOG1), hwnd,
AboutDlgProc);

  // Enter main event loop
```

```
    while (true)
    {
      // Test if there is a message in queue, if so get it.
      if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
        // Test if this is a quit.
        if (msg.message == WM_QUIT)
        {
          break;
        }

        // Translate any accelerator keys.
        TranslateMessage(&msg);
        // Send the message to the window proc.
        DispatchMessage(&msg);
      }

      // Main game processing goes here.
      GameLoop(); //One frame of game logic occurs here...
    }

    // Return to Windows like this...
    return (static_cast<int>(msg.wParam));
  }
```

How it works...

After the `resource.h` file is automatically created for us, we can manually edit it to name the dialog appropriately. After the main window is created, we need to get a handle to the window and then call the dialog box function like this:

```
DialogBox(_hInstance, MAKEINTRESOURCE(IDD_DIALOG1), hwnd,
AboutDlgProc)
```

Very much like the main window callback, the dialog box has its own callback. We need to intercept the messages accordingly and perform our actions. `BOOL CALLBACK AboutDlgProc` is the callback that we have at our disposal. We have a similar initialize message. For our dialog boxes, most of the intercepts will take place in `WM_COMMAND`. Based on the `wparam` parameter, we need to switch, so that know whether we have clicked the **OK** button or the **CANCEL** button and can take the appropriate steps.

Using sprites

To develop any 2D game, we need sprites. Sprites are elements of computer graphics that can stay on screen, be manipulated, and be animated. GDI allows us to use sprites to create our game. Probably all the assets in the game will be sprites, from the UI to the main characters, and so on.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how to use sprites in our game:

1. Open Visual Studio.
2. Create a new C++ project.
3. Create a new resource type.
4. Select the **Sprite** option as the new resource type.
5. Add the following source files: `backbuffer.h/cpp`, `Clock.h/cpp`, `Game.h/.cpp`, `sprite.h/cpp`, and `Utilities.h`.
6. Add the following lines of code to `backbuffer.h`:

```
#pragma once

#if !defined(__BACKBUFFER_H__)
#define __BACKBUFFER_H__

// Library Includes
#include <Windows.h>

// Local Includes

// Types

// Constants

// Prototypes
class CBackBuffer
{
  // Member Functions
public:
```

```
  CBackBuffer();
  ~CBackBuffer();

  bool Initialise(HWND _hWnd, int _iWidth, int _iHeight);

  HDC GetBFDC() const;

  int GetHeight() const;
  int GetWidth() const;

  void Clear();
  void Present();

protected:

private:
  CBackBuffer(const CBackBuffer& _kr);
  CBackBuffer& operator= (const CBackBuffer& _kr);

  // Member Variables
public:

protected:
  HWND m_hWnd;
  HDC m_hDC;
  HBITMAP m_hSurface;
  HBITMAP m_hOldObject;
  int m_iWidth;
  int m_iHeight;

private:

};

#endif    // __BACKBUFFER_H__
```

7. Add the following lines of code to `backbuffer.cpp`:

```
// Library Includes

// Local Includes

// This include
```

```
#include "BackBuffer.h"

// Static Variables

// Static Function Prototypes

// Implementation

CBackBuffer::CBackBuffer()
: m_hWnd(0)
, m_hDC(0)
, m_hSurface(0)
, m_hOldObject(0)
, m_iWidth(0)
, m_iHeight(0)
{

}

CBackBuffer::~CBackBuffer()
{
  SelectObject(m_hDC, m_hOldObject);

  DeleteObject(m_hSurface);
  DeleteObject(m_hDC);
}

bool
CBackBuffer::Initialise(HWND _hWnd, int _iWidth, int _iHeight)
{
  m_hWnd = _hWnd;

  m_iWidth = _iWidth;
  m_iHeight = _iHeight;

  HDC hWindowDC = ::GetDC(m_hWnd);

  m_hDC = CreateCompatibleDC(hWindowDC);

  m_hSurface = CreateCompatibleBitmap(hWindowDC, m_iWidth, m_
iHeight);

  ReleaseDC(m_hWnd, hWindowDC);

  m_hOldObject = static_cast<HBITMAP>(SelectObject(m_hDC, m_
```

```
hSurface));

  HBRUSH brushWhite = static_cast<HBRUSH>(GetStockObject(LTGRAY_
BRUSH));
  HBRUSH oldBrush = static_cast<HBRUSH>(SelectObject(m_hDC,
brushWhite));

  Rectangle(m_hDC, 0, 0, m_iWidth, m_iHeight);

  SelectObject(m_hDC, oldBrush);

  return (true);
}

void
CBackBuffer::Clear()
{
  HBRUSH hOldBrush = static_cast<HBRUSH>(SelectObject(GetBFDC(),
GetStockObject(LTGRAY_BRUSH)));

  Rectangle(GetBFDC(), 0, 0, GetWidth(), GetHeight());

  SelectObject(GetBFDC(), hOldBrush);
}

HDC
CBackBuffer::GetBFDC() const
{
  return (m_hDC);
}

int
CBackBuffer::GetWidth() const
{
  return (m_iWidth);
}

int
CBackBuffer::GetHeight() const
{
  return (m_iHeight);
}

void
```

```
CBackBuffer::Present()
{
  HDC hWndDC = ::GetDC(m_hWnd);

  BitBlt(hWndDC, 0, 0, m_iWidth, m_iHeight, m_hDC, 0, 0, SRCCOPY);

  ReleaseDC(m_hWnd, hWndDC);
}
```

8. Add the following lines of code to `Clock.h`:

```
#pragma once

#if !defined(__CLOCK_H__)
#define __CLOCK_H__

// Library Includes

// Local Includes

// Types

// Constants

// Prototypes
class CClock
{
  // Member Functions
public:
  CClock();
  ~CClock();

  bool Initialise();

  void Process();

  float GetDeltaTick();

protected:

private:
  CClock(const CClock& _kr);
  CClock& operator= (const CClock& _kr);

  // Member Variables
```

```
public:

protected:
  float m_fTimeElapsed;
  float m_fDeltaTime;
  float m_fLastTime;
  float m_fCurrentTime;

private:

};

#endif    // __CLOCK_H__
```

9. Add the following lines of code to `Clock.cpp`:

```
// Library Includes
#include <windows.h>

// Local Includes
#include "Clock.h"

// Static Variables

// Static Function Prototypes

// Implementation

CClock::CClock()
: m_fTimeElapsed(0.0f)
, m_fDeltaTime(0.0f)
, m_fLastTime(0.0f)
, m_fCurrentTime(0.0f)
{

}

CClock::~CClock()
{

}

bool
CClock::Initialise()
{
```

```
  return (true);
}

void
CClock::Process()
{
  m_fLastTime = m_fCurrentTime;

  m_fCurrentTime = static_cast<float>(timeGetTime());

  if (m_fLastTime == 0.0f)
  {
    m_fLastTime = m_fCurrentTime;
  }

  m_fDeltaTime = m_fCurrentTime - m_fLastTime;

  m_fTimeElapsed += m_fDeltaTime;
}

float
CClock::GetDeltaTick()
{
  return (m_fDeltaTime / 1000.0f);
}
```

10. Add the following lines of code to `Game.h`:

```
#pragma once

#if !defined(__GAME_H__)
#define __GAME_H__

// Library Includes
#include <windows.h>

// Local Includes
#include "clock.h"

// Types

// Constants

// Prototypes
```

```
class CBackBuffer;

class CGame
{
  // Member Functions
public:
  ~CGame();

  bool Initialise(HINSTANCE _hInstance, HWND _hWnd, int _iWidth,
int _iHeight);

  void Draw();
  void Process(float _fDeltaTick);

  void ExecuteOneFrame();

  CBackBuffer* GetBackBuffer();
  HINSTANCE GetAppInstance();
  HWND GetWindow();

  // Singleton Methods
  static CGame& GetInstance();
  static void DestroyInstance();

protected:

private:
  CGame();
  CGame(const CGame& _kr);
  CGame& operator= (const CGame& _kr);

  // Member Variables
public:

protected:
  CClock* m_pClock;

  CBackBuffer* m_pBackBuffer;

  //Application data
  HINSTANCE m_hApplicationInstance;
  HWND m_hMainWindow;

  // Singleton Instance
```

```
    static CGame* s_pGame;

private:

};

#endif    // __GAME_H__
```

11. Add the following lines of code to `Game.cpp`:

```
// Library Includes

// Local Includes
#include "Clock.h"
#include "BackBuffer.h"
#include "Utilities.h"

// This Include
#include "Game.h"

// Static Variables
CGame* CGame::s_pGame = 0;

// Static Function Prototypes

// Implementation

CGame::CGame()
: m_pClock(0)
, m_hApplicationInstance(0)
, m_hMainWindow(0)
, m_pBackBuffer(0)
{

}

CGame::~CGame()
{
  delete m_pBackBuffer;
  m_pBackBuffer = 0;

  delete m_pClock;
```

```
  m_pClock = 0;
}

bool
CGame::Initialise(HINSTANCE _hInstance, HWND _hWnd, int _iWidth,
int _iHeight)
{
  m_hApplicationInstance = _hInstance;
  m_hMainWindow = _hWnd;

  m_pClock = new CClock();
  VALIDATE(m_pClock->Initialise());
  m_pClock->Process();

  m_pBackBuffer = new CBackBuffer();
  VALIDATE(m_pBackBuffer->Initialise(_hWnd, _iWidth, _iHeight));

  ShowCursor(false);

  return (true);
}

void
CGame::Draw()
{
  m_pBackBuffer->Clear();

  // Do all the game's drawing here...

  m_pBackBuffer->Present();
}

void
CGame::Process(float _fDeltaTick)
{
  // Process all the game's logic here.
}

void
CGame::ExecuteOneFrame()
{
  float fDT = m_pClock->GetDeltaTick();

  Process(fDT);
```

```
  Draw();

  m_pClock->Process();

  Sleep(1);
}

CGame&
CGame::GetInstance()
{
  if (s_pGame == 0)
  {
    s_pGame = new CGame();
  }

  return (*s_pGame);
}

void
CGame::DestroyInstance()
{
  delete s_pGame;
  s_pGame = 0;
}

CBackBuffer*
CGame::GetBackBuffer()
{
  return (m_pBackBuffer);
}

HINSTANCE
CGame::GetAppInstance()
{
  return (m_hApplicationInstance);
}

HWND
CGame::GetWindow()
{
  return (m_hMainWindow);
}
```

12. Add the following lines of code to `sprite.h`:

```
#pragma once

#if !defined(__SPRITE_H__)
#define __SPRITE_H__

// Library Includes
#include "windows.h"

// Local Includes

// Types

// Constants

// Prototypes
class CSprite
{
  // Member Functions
public:
  CSprite();
  ~CSprite();

  bool Initialise(int _iResourceID, int _iMaskResourceID);

  void Draw();
  void Process(float _fDeltaTick);

  int GetWidth() const;
  int GetHeight() const;

  int GetX() const;
  int GetY() const;
  void SetX(int _i);
  void SetY(int _i);

  void TranslateRelative(int _iX, int _iY);
  void TranslateAbsolute(int _iX, int _iY);

protected:

private:
  CSprite(const CSprite& _kr);
```

```
  CSprite& operator= (const CSprite& _kr);

  // Member Variables
public:

protected:
  //Center handle
  int m_iX;
  int m_iY;

  HBITMAP m_hSprite;
  HBITMAP m_hMask;

  BITMAP m_bitmapSprite;
  BITMAP m_bitmapMask;

  static HDC s_hSharedSpriteDC;
  static int s_iRefCount;

private:

};

#endif    // __SPRITE_H__
```

13. Add the following lines of code to `sprite.cpp`:

```
// Library Includes

// Local Includes
#include "resource.h"
#include "Game.h"
#include "BackBuffer.h"
#include "Utilities.h"

// This include
#include "Sprite.h"

// Static Variables
HDC CSprite::s_hSharedSpriteDC = 0;
int CSprite::s_iRefCount = 0;

// Static Function Prototypes
```

```
// Implementation

CSprite::CSprite()
: m_iX(0)
, m_iY(0)
{
  ++s_iRefCount;
}

CSprite::~CSprite()
{
  DeleteObject(m_hSprite);
  DeleteObject(m_hMask);

  --s_iRefCount;

  if (s_iRefCount == 0)
  {
    DeleteDC(s_hSharedSpriteDC);
    s_hSharedSpriteDC = 0;
  }
}

bool
CSprite::Initialise(int _iSpriteResourceID, int _iMaskResourceID)
{
  HINSTANCE hInstance = CGame::GetInstance().GetAppInstance();

  if (!s_hSharedSpriteDC)
  {
    s_hSharedSpriteDC = CreateCompatibleDC(NULL);
  }

  m_hSprite = LoadBitmap(hInstance, MAKEINTRESOURCE(_
iSpriteResourceID));
  VALIDATE(m_hSprite);
  m_hMask = LoadBitmap(hInstance, MAKEINTRESOURCE(_
iMaskResourceID));
  VALIDATE(m_hMask);

  GetObject(m_hSprite, sizeof(BITMAP), &m_bitmapSprite);
  GetObject(m_hMask, sizeof(BITMAP), &m_bitmapMask);

  return (true);
```

```
}

void
CSprite::Draw()
{
  int iW = GetWidth();
  int iH = GetHeight();

  int iX = m_iX - (iW / 2);
  int iY = m_iY - (iH / 2);

  CBackBuffer* pBackBuffer = CGame::GetInstance().GetBackBuffer();

  HGDIOBJ hOldObj = SelectObject(s_hSharedSpriteDC, m_hMask);

  BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s_
hSharedSpriteDC, 0, 0, SRCAND);

  SelectObject(s_hSharedSpriteDC, m_hSprite);

  BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s_
hSharedSpriteDC, 0, 0, SRCPAINT);

  SelectObject(s_hSharedSpriteDC, hOldObj);
}

void
CSprite::Process(float _fDeltaTick)
{

}

int
CSprite::GetWidth() const
{
  return (m_bitmapSprite.bmWidth);
}

int
CSprite::GetHeight() const
{
```

```
  return (m_bitmapSprite.bmHeight);
}

int
CSprite::GetX() const
{
  return (m_iX);
}

int
CSprite::GetY() const
{
  return (m_iY);
}

void
CSprite::SetX(int _i)
{
  m_iX = _i;
}

void
CSprite::SetY(int _i)
{
  m_iY = _i;
}

void
CSprite::TranslateRelative(int _iX, int _iY)
{
  m_iX += _iX;
  m_iY += _iY;
}

void
CSprite::TranslateAbsolute(int _iX, int _iY)
{
  m_iX = _iX;
  m_iY = _iY;
}
```

14. Add the following lines of code to `Utilities.h`:

```
// Library Includes

// Local Includes
#include "resource.h"
#include "Game.h"
#include "BackBuffer.h"
#include "Utilities.h"

// This include
#include "Sprite.h"

// Static Variables
HDC CSprite::s_hSharedSpriteDC = 0;
int CSprite::s_iRefCount = 0;

// Static Function Prototypes

// Implementation

CSprite::CSprite()
: m_iX(0)
, m_iY(0)
{
  ++s_iRefCount;
}

CSprite::~CSprite()
{
  DeleteObject(m_hSprite);
  DeleteObject(m_hMask);

  --s_iRefCount;

  if (s_iRefCount == 0)
  {
    DeleteDC(s_hSharedSpriteDC);
    s_hSharedSpriteDC = 0;
  }
}

bool
CSprite::Initialise(int _iSpriteResourceID, int _iMaskResourceID)
{
```

```
  HINSTANCE hInstance = CGame::GetInstance().GetAppInstance();

  if (!s_hSharedSpriteDC)
  {
    s_hSharedSpriteDC = CreateCompatibleDC(NULL);
  }

  m_hSprite = LoadBitmap(hInstance, MAKEINTRESOURCE(_
iSpriteResourceID));
  VALIDATE(m_hSprite);
  m_hMask = LoadBitmap(hInstance, MAKEINTRESOURCE(_
iMaskResourceID));
  VALIDATE(m_hMask);

  GetObject(m_hSprite, sizeof(BITMAP), &m_bitmapSprite);
  GetObject(m_hMask, sizeof(BITMAP), &m_bitmapMask);

  return (true);
}

void
CSprite::Draw()
{
  int iW = GetWidth();
  int iH = GetHeight();

  int iX = m_iX - (iW / 2);
  int iY = m_iY - (iH / 2);

  CBackBuffer* pBackBuffer = CGame::GetInstance().GetBackBuffer();

  HGDIOBJ hOldObj = SelectObject(s_hSharedSpriteDC, m_hMask);

  BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s_
hSharedSpriteDC, 0, 0, SRCAND);

  SelectObject(s_hSharedSpriteDC, m_hSprite);

  BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s_
hSharedSpriteDC, 0, 0, SRCPAINT);

  SelectObject(s_hSharedSpriteDC, hOldObj);
}

void
```

```
CSprite::Process(float _fDeltaTick)
{

}

int
CSprite::GetWidth() const
{
  return (m_bitmapSprite.bmWidth);
}

int
CSprite::GetHeight() const
{
  return (m_bitmapSprite.bmHeight);
}

int
CSprite::GetX() const
{
  return (m_iX);
}

int
CSprite::GetY() const
{
  return (m_iY);
}

void
CSprite::SetX(int _i)
{
  m_iX = _i;
}

void
CSprite::SetY(int _i)
{
  m_iY = _i;
}

void
CSprite::TranslateRelative(int _iX, int _iY)
{
```

```
    m_iX += _iX;
    m_iY += _iY;
  }

  void
  CSprite::TranslateAbsolute(int _iX, int _iY)
  {
    m_iX = _iX;
    m_iY = _iY;
  }
```

How it works...

As we know, the backbuffer is used to draw the image first and then we swap the buffer to present it to the screen. This process is also called *presenting*. We create a generic `backbuffer` class that helps us to swap the buffer. The `sprite` class is used to load the sprites and push them on to the back buffer, where they can be processed and finally drawn on the screen. The sprite class is also provided with some basic utility functions that help us to get the width and height of the sprite. Most of the functions are just a wrapper on top of Windows' own API functions and callbacks. We have also created a `clock` class, which helps us to keep track of time, as each time point should be implemented as a function of delta time. If we do not do this, then the game might run with fluctuating behavior, based on the machine that is executing it. The `game` class is used to put all of it together. It has an instance of `backbuffer`, which is a singleton class and handles the context of the window and other resources.

Using animated sprites

Using animated sprites is an important part of games programming. Unless some kind of animation is applied to the sprites, it will not appear realistic enough and the whole mood of immersion in the game will be lost. Although animations can be achieved in a variety of ways, we will only look at sprite strip animation, as it is the most commonly used form of animation for 2D games.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to create dialog boxes.

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.
4. Add a `AnimatedSprite.cpp` file.
5. Add the following lines of code to `Source.cpp`:

```
// This include
#include "AnimatedSprite.h"

// Static Variables

// Static Function Prototypes

// Implementation

CAnimatedSprite::CAnimatedSprite()
: m_fFrameSpeed(0.0f)
, m_fTimeElapsed(0.0f)
, m_iCurrentSprite(0)
{

}

CAnimatedSprite::~CAnimatedSprite()
{
  Deinitialise();
}

bool
CAnimatedSprite::Deinitialise()
{
  return (CSprite::Deinitialise());
}

bool
CAnimatedSprite::Initialise(int _iSpriteResourceID, int _
iMaskResourceID)
{
  return (CSprite::Initialise(_iSpriteResourceID, _
iMaskResourceID));
```

```
}

void
CAnimatedSprite::Draw()
{
  int iTopLeftX = m_vectorFrames[m_iCurrentSprite];
  int iTopLeftY = 0;

  int iW = GetFrameWidth();
  int iH = GetHeight();

  int iX = m_iX - (iW / 2);
  int iY = m_iY - (iH / 2);

  HDC hSpriteDC = hSharedSpriteDC;

  HGDIOBJ hOldObj = SelectObject(hSpriteDC, m_hMask);

  BitBlt(CGame::GetInstance().GetBackBuffer()->GetBFDC(), iX, iY,
iW, iH, hSpriteDC, iTopLeftX, iTopLeftY, SRCAND);

  SelectObject(hSpriteDC, m_hSprite);

  BitBlt(CGame::GetInstance().GetBackBuffer()->GetBFDC(), iX, iY,
iW, iH, hSpriteDC, iTopLeftX, iTopLeftY, SRCPAINT);

  SelectObject(hSpriteDC, hOldObj);
}

void
CAnimatedSprite::Process(float _fDeltaTick)
{
  m_fTimeElapsed += _fDeltaTick;

  if (m_fTimeElapsed >= m_fFrameSpeed &&
    m_fFrameSpeed != 0.0f)
  {
    m_fTimeElapsed = 0.0f;
    ++m_iCurrentSprite;

    if (m_iCurrentSprite >= m_vectorFrames.size())
    {
      m_iCurrentSprite = 0;
    }
```

```
    }

    CSprite::Process(_fDeltaTick);
  }

  void
  CAnimatedSprite::AddFrame(int _iX)
  {
    m_vectorFrames.push_back(_iX);
  }

  void
  CAnimatedSprite::SetSpeed(float _fSpeed)
  {
    m_fFrameSpeed = _fSpeed;
  }

  void
  CAnimatedSprite::SetWidth(int _iW)
  {
    m_iFrameWidth = _iW;
  }

  int
  CAnimatedSprite::GetFrameWidth()
  {
    return (m_iFrameWidth);
  }
```

How it works...

For the animation to work, we need to load in a sequence of images as sprite strips. The higher the number of images, the smoother the animation will be. For the equivalent number of sprites, we need to load in their masks as well, so that they can be blitted together. We need to store all the images in a vector list. For the animation to work properly, all the images must be equally spaced out. After we have stored them correctly, we can run the animations as rapidly or slowly as we want, by controlling how many frames/sprites we want to draw in a certain amount of time. The remaining process of drawing the sprite on the screen remains the same.

6
Design Patterns for Game Development

In this chapter, the following recipes will be covered:

- Using the singleton design pattern
- Using the factory method
- Using the abstract factory method
- Using the observer pattern
- Using the flyweight pattern
- Using the strategy pattern
- Using the command design pattern
- Creating an advanced game using design patterns

Introduction

Let us consider that we are faced with a certain problem. After some time, we find a solution to that problem. Now, if the problem reoccurs, or a similar pattern to the problem reoccurs, we will know how to solve the problem by applying the same principle that solved the previous problem. Design patterns are similar to this. There are already 23 such solutions documented, which provide subtle solutions for dealing with problems that have a similar pattern to the ones that are documented. They are described by the authors more commonly referred to as the *Gang of Four*. They are not complete solutions, but rather templates or frameworks that can be applied to similar situations. One of the biggest drawbacks of design patterns, however, is that if they are not applied correctly, they can prove to be disastrous. Design patterns can be classified as structural, behavioral, or creational. We will be looking at only a few of them, which are used often in games development.

Using the singleton design pattern

The singleton design pattern is the most commonly used design pattern for games. Unfortunately, it is also the most overused and most incorrectly applied design pattern for games. There are a few advantages of the singleton design pattern, which we will discuss. However, it has a lot of serious consequences as well.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to create a singleton design pattern. We will also see the common pitfalls of this design pattern:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 console application.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;

class PhysicsManager
{
private:
  static bool bCheckFlag;
  static PhysicsManager *s_singleInstance;
  PhysicsManager()
  {
    //private constructor
  }
public:
  static PhysicsManager* getInstance();
  void GetCurrentGravity()const;

  ~PhysicsManager()
```

```
  {
    bCheckFlag = false;
  }
};

bool PhysicsManager::bCheckFlag = false;

PhysicsManager* PhysicsManager::s_singleInstance = NULL;

PhysicsManager* PhysicsManager::getInstance()
{
  if (!bCheckFlag)
  {
    s_singleInstance = new PhysicsManager();
    bCheckFlag = true;
    return s_singleInstance;
  }
  else
  {
    return s_singleInstance;
  }
}

void PhysicsManager::GetCurrentGravity() const
{
  //Some calculations for finding the current gravity
  //Probably a base variable which constantly gets updated with
value
  //based on the environment
  cout << "Current gravity of the system is: " <<9.8<< endl;
}

int main()
{
  PhysicsManager *sc1, *sc2;
  sc1 = PhysicsManager::getInstance();
  sc1->GetCurrentGravity();
  sc2 = PhysicsManager::getInstance();
  sc2->GetCurrentGravity();

  _getch();
  return 0;
}
```

How it works...

The main reason why developers want to use a singleton class is when they want to restrict to just one instance of the class. In our example, we have taken the `PhysicsManager` class. We make the constructor private and then assign a static function to get the handle to the instance of the class and hence its methods. We also use a Boolean to check if an instance is already created. If it is, we do not assign a new instance. If it is not, we assign a new instance and call the corresponding methods.

As intelligent as it may seem, this design pattern has many flaws and hence should be avoided as much as possible in game design. First, it's a global variable. This in itself is bad. A global variable is saved on the global pool and can be accessed from everywhere. Second, this encourages bad coupling, which may appear in the code. Third, it is not concurrent friendly. Imagine there are multiple threads, and each thread can access this global variable. This is a recipe for disaster, as deadlock will happen. Finally, one of the most common mistakes made by new programmers is to create managers for everything, and then make the manager a singleton. The fact is that we can get away without creating a manager by using OOPS and references in an effective manner.

The preceding code shows a lazy value of initializing a singleton and hence can be improved. However, all the fundamental problems described in this recipe will still remain.

Using the factory method

A factory is essentially a warehouse for creating objects of other types. In a factory method design pattern, the creation of a new type of object, such as an enemy or a building, happens from an interface and the subclass decides which class it needs to instantiate. This is also a commonly used pattern in games and can be quite useful.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how easy it is to write a factory method design pattern:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 console application.
4. Add a source file called `Source.cpp`.

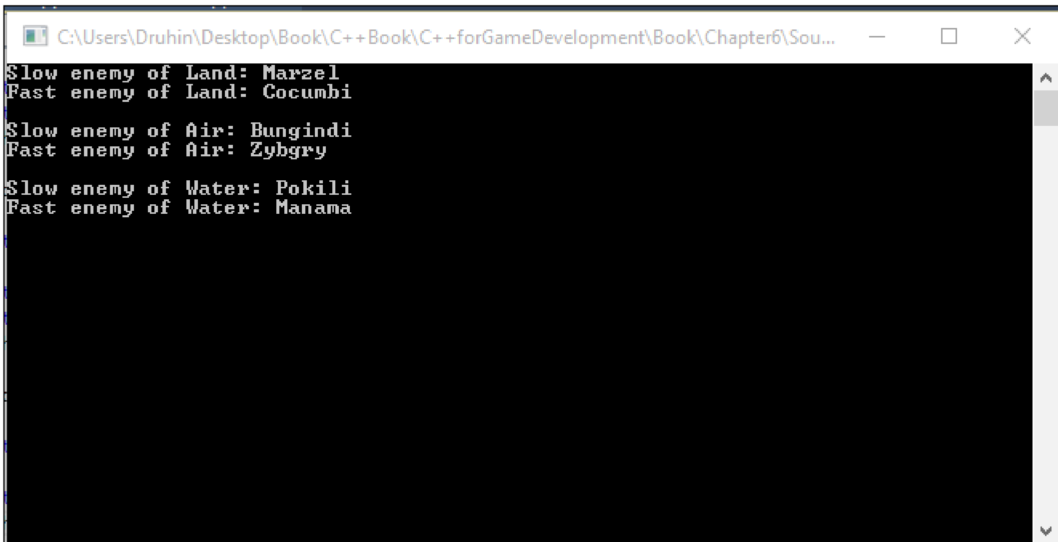
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>
#include <vector>

using namespace std;

class IBuilding
{
public:
  virtual void TotalHealth() = 0;
};

class Barracks : public IBuilding
{
public:
  void TotalHealth()
  {
    cout << "Health of Barrack is :" << 100;
  }
};
class Temple : public IBuilding
{
public:
  void TotalHealth()
  {
    cout << "Health of Temple is :" << 75;
  }
};
class Farmhouse : public IBuilding
{
public:
  void TotalHealth()
  {
    cout << "Health of Farmhouse is :" << 50;
  }
};

int main()
{
  vector<IBuilding*> BuildingTypes;
```

```
    int choice;

    cout << "Specify the different building types in your village"
  << endl;
    while (true)
    {

      cout << "Barracks(1) Temple(2) Farmhouse(3) Go(0): ";
      cin >> choice;
      if (choice == 0)
        break;
      else if (choice == 1)
        BuildingTypes.push_back(new Barracks);
      else if (choice == 2)
        BuildingTypes.push_back(new Temple);
      else
        BuildingTypes.push_back(new Farmhouse);
    }
    cout << endl;
    cout << "There are total " << BuildingTypes.size() << "
  buildings" << endl;
    for (int i = 0; i < BuildingTypes.size(); i++)
    {
      BuildingTypes[i]->TotalHealth();
      cout << endl;
    }

    for (int i = 0; i < BuildingTypes.size(); i++)
      delete BuildingTypes[i];

    _getch();
  }
```

How it works...

In this example, we have created a `Building` interface, which has a pure virtual function for `TotalHealth`. This means all the derived classes must override this function. Therefore, we can guarantee that all our buildings will have this property. We can keep adding to this structure by having more properties, such as hit points, total storage capacity, rate of production of villagers, and so on, based on the nature and design of the game. The derived classes have their own implementation of `TotalHealth`. They are also named to reflect the type of building they are. The biggest advantage of this design pattern is that all we need on the client side is a reference to the base interface. After that, we can create the type of building we need at runtime. We store these building types in a vector list and finally use a loop to display the contents. Since we have the reference `IBuilding*`, we can assign any new derived class we want at runtime. There is no need to create references for all derived classes, such as `Temple*` and so on. The following screenshot shows the output we are likely to get for a user-defined village:



Using the abstract factory method

An abstract factory is a part of the creational design pattern. It is one of the best ways to create an object and is a commonly repeated design pattern in games. It is like a factory of factories. It uses an interface to create a factory. The factory is responsible for creating objects without specifying their class type. The factory generates these objects based on the factory method design pattern. However, some argue that the abstract factory method can also be implemented using the prototype design pattern.

Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

How to do it...

In this recipe, we will find out how easy it is to implement the abstract factory pattern:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 console application.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>
#include <string>

using namespace std;

//IFast interface
class IFast
{
public:
  virtual std::string Name() = 0;
};

//ISlow interface
class ISlow
{
public:
  virtual std::string Name() = 0;
};
class Rapter : public ISlow
{
public:
  std::string Name()
  {
    return "Rapter";
  }
};

class Cocumbi : public IFast
```

```
{
public:
  std::string Name()
  {
    return "Cocumbi";
  }
};
   . . . . .// Similar classes can be written here
class AEnemyFactory
{
public:
  enum Enemy_Factories
  {
    Land,
    Air,
    Water
  };

  virtual IFast* GetFast() = 0;
  virtual ISlow* GetSlow() = 0;

  static AEnemyFactory* CreateFactory(Enemy_Factories factory);
};

class LandFactory : public AEnemyFactory
{
public:
  IFast* GetFast()
  {
    return new Cocumbi();
  }

  ISlow* GetSlow()
  {
    return new Marzel();
  }
};

class AirFactory : public AEnemyFactory
{
public:
  IFast* GetFast()
  {
    return new Zybgry();
```

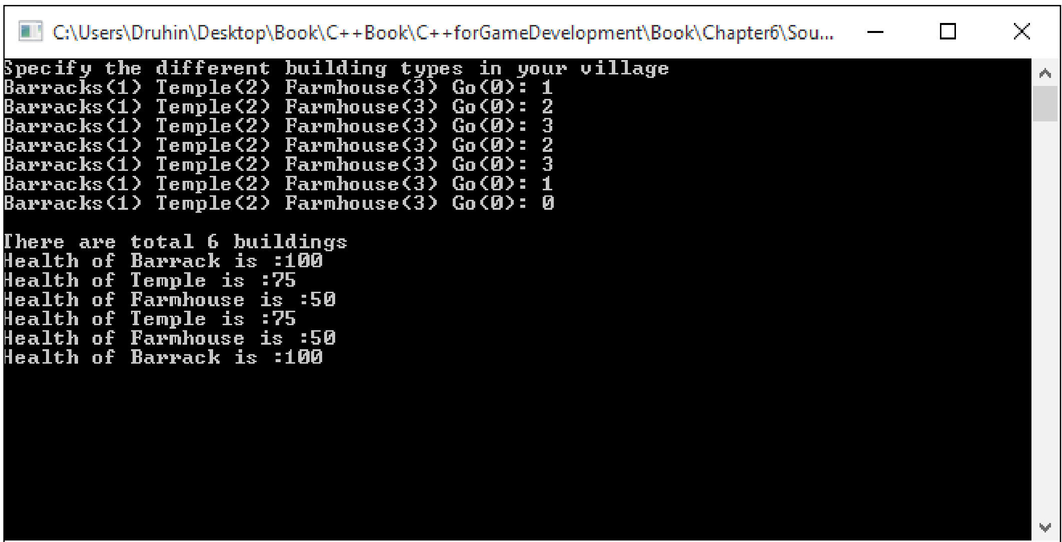
```
    }

    ISlow* GetSlow()
    {
      return new Bungindi();
    }
  };

  class WaterFactory : public AEnemyFactory
  {
  public:
    IFast* GetFast()
    {
      return new Manama();
    }

    ISlow* GetSlow()
    {
      return new Pokili();
    }
  };

  //CPP File
  AEnemyFactory* AEnemyFactory::CreateFactory(Enemy_Factories
  factory)
  {
    if (factory == Enemy_Factories::Land)
    {
      return new LandFactory();
    }
    else if (factory == Enemy_Factories::Air)
    {
      return new AirFactory();
    }
    else if (factory == Enemy_Factories::Water)
    {
      return new WaterFactory();
    }
  }

  int main(int argc, char* argv[])
  {
    AEnemyFactory *factory = AEnemyFactory::CreateFactory
```

```
    (AEnemyFactory::Enemy_Factories::Land);

  cout << "Slow enemy of Land: " << factory->GetSlow()->Name() <<
"\n";
  delete factory->GetSlow();
  cout << "Fast enemy of Land: " << factory->GetFast()->Name() <<
"\n";
  delete factory->GetFast();
  delete factory;
  getchar();

  factory = AEnemyFactory::CreateFactory(AEnemyFactory::Enemy_
Factories::Air);
  cout << "Slow enemy of Air: " << factory->GetSlow()->Name() <<
"\n";
  delete factory->GetSlow();
  cout << "Fast enemy of Air: " << factory->GetFast()->Name() <<
"\n";
  delete factory->GetFast();
  delete factory;
  getchar();

  factory = AEnemyFactory::CreateFactory(AEnemyFactory::Enemy_
Factories::Water);
  cout << "Slow enemy of Water: " << factory->GetSlow()->Name() <<
"\n";
  delete factory->GetSlow();
  cout << "Fast enemy of Water: " << factory->GetFast()->Name() <<
"\n";
  delete factory->GetFast();
  getchar();

  return 0;
}
```

How it works...

In this example, we have created two interfaces, namely `IFast` and `ISlow`. After that we have created several enemies and decided whether they are fast or slow. Finally, we created an abstract class with two virtual functions to get the fast enemy and the slow enemy. This means all the derived classes must override and have their own implementation of these functions. So in effect we have created a factory of factories. The land, air, and water enemy factories that we have created from the abstract class have references to two interfaces for slow and fast. Hence the land, water, and air serve as factories themselves too.

So from the client side, we can request a fast land enemy or a slow water enemy and we can get the appropriate enemy displayed to us. As the following screenshot shows, we can get the output as displayed:



Using the observer pattern

The observer design pattern is one which is not commonly used in games, but it should be used more often by game developers as it is a very smart way to handle notifications. In the observer design pattern, a component maintains a one-to-many relationship with other components. This means when the main component changes, all the dependent components also update. Imagine a physics system. We want `enemy1` and `enemy2` to update as soon as the physics system updates, so we should use this pattern.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to implement the observer pattern:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 Windows application.

4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <vector>
#include <conio.h>

using namespace std;

class PhysicsSystem {

  vector < class Observer * > views;
  int value;
public:
  void attach(Observer *obs) {
    views.push_back(obs);
  }
  void setVal(int val) {
    value = val;
    notify();
  }
  int getVal() {
    return value;
  }
  void notify();
};

class Observer {

  PhysicsSystem *_attribute;
  int iScalarMultiplier;
public:
  Observer(PhysicsSystem *attribute, int value)
  {
    If(attribute)
{

_attribute = attribute;
}
    iScalarMultiplier = value;

    _attribute->attach(this);
  }
  virtual void update() = 0;
```

```
protected:
  PhysicsSystem *getPhysicsSystem() {
    return _attribute;
  }
  int getvalue()
  {
    return iScalarMultiplier;
  }
};

void PhysicsSystem::notify() {

  for (int i = 0; i < views.size(); i++)
    views[i]->update();
}

class PlayerObserver : public Observer {
public:
  PlayerObserver(PhysicsSystem *attribute, int value) :
Observer(attribute, value){}
  void update() {

    int v = getPhysicsSystem()->getVal(), d = getvalue();
    cout << "Player is dependent on the Physics system" << endl;
    cout << "Player new impulse value is " << v / d << endl <<
endl;
  }
};

class AIObserver : public Observer {
public:
  AIObserver(PhysicsSystem *attribute, int value) :
Observer(attribute, value){}
  void update() {
    int v = getPhysicsSystem()->getVal(), d = getvalue();
    cout << "AI is dependent on the Physics system" << endl;
    cout << "AI new impulse value is " << v % d << endl << endl;
  }
};

int main() {
  PhysicsSystem subj;

  PlayerObserver valueObs1(&subj, 4);
```

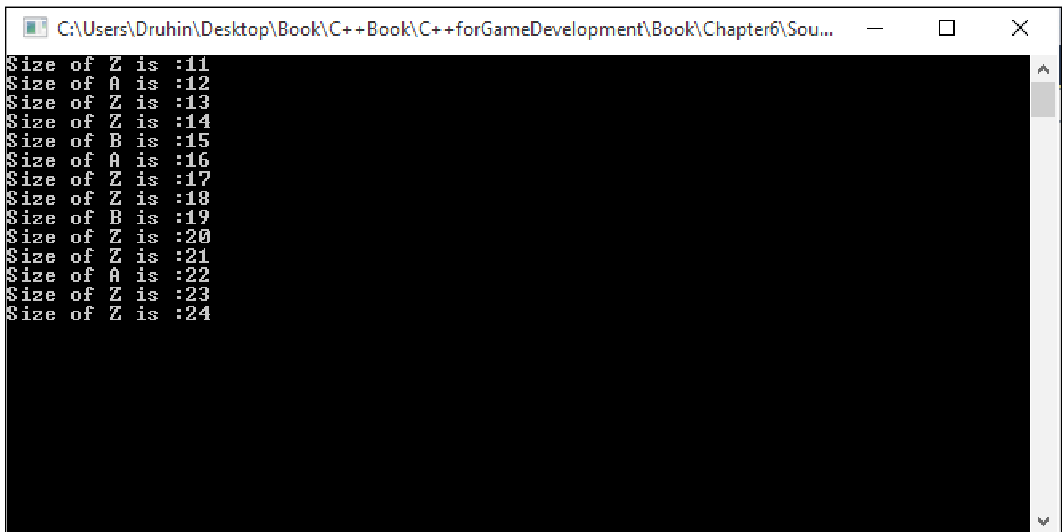
```
    AIObserver attributeObs3(&subj, 3);
    subj.setVal(100);

    _getch();
}
```

How it works...

In this example, we have created a physics system that continuously updates its value. Other components that are dependent on the physics system must attach themselves to it, so that they are notified as soon as the physics system is updated.

The physics system that we have created holds a vector list of all the components that are being observing from it. In addition to this, it contains methods to get the current value or set a values for it. It also contains a method to notify all the dependent components once a value has been changed in the physics system. The `Observer` class contains a reference to the physics system, as well as a pure virtual function for updates, which the derived class must override. The `PlayerObserver` and `AIObserver` classes can derive from this class and have their own implementation of impulse based on the changes in the physics system. Both the AI and player systems will continuously receive updates from the physics system unless they detach themselves from it.

This is a very useful pattern and has loads of implementation in games. The following screenshot shows what a typical output would look like:



Using the flyweight pattern

The flyweight design pattern is mostly used when we want to reduce the amount of memory that is used to create the objects. This pattern is often used when we want to create something hundreds or thousands of times. Games with a forest structure often use this design pattern. This design pattern falls under the structural design category. In this pattern, the object, let's say the tree object, is divided into two parts, one that is dependent on the state of the object and one that is independent. The independent part is stored in the flyweight object, whereas the dependent part is handled by the client and sent to the flyweight object as and when invoked.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to implement the flyweight pattern:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 console application.
4. Add a source file called `Source.cpp`.
5. Add the following lines of code to it:

```
#include <iostream>
#include <string>
#include <map>
#include <conio.h>

using namespace std;


class TreeType
{
public:
  virtual void Display(int size) = 0;

protected:
  //Some Model we need to assign. For relevance we are
substituting this with a character symbol
  char symbol_;
```

```
  int width_;
  int height_;
  float color_;

  int Size_;
};


class TreeTypeA : public TreeType
{
public:
  TreeTypeA()
  {
    symbol_ = 'A';
    width_ = 94;
    height_ = 135;
    color_ = 0;

    Size_ = 0;
  }
  void Display(int size)
  {
    Size_ = size;
    cout << "Size of " << symbol_ << " is :" << Size_ << endl;
  }
};


class TreeTypeB : public TreeType
{
public:
  TreeTypeB()
  {
    symbol_ = 'B';
    width_ = 70;
    height_ = 25;
    color_ = 0;

    Size_ = 0;
  }
  void Display(int size)
  {
    Size_ = size;
```

```
      cout << "Size of " << symbol_ << " is :" << Size_ << endl;
    }
  };

  class TreeTypeZ : public TreeType
  {
  public:
    TreeTypeZ()
    {
      symbol_ = 'Z';
      width_ = 20;
      height_ = 40;
      color_ = 1;

      Size_ = 0;
    }
    void Display(int size)
    {
      Size_ = size;
      cout <<"Size of " << symbol_ << " is :" << Size_ << endl;
    }
  };

  // The 'FlyweightFactory' class
  class TreeTypeFactory
  {
  public:
    virtual ~TreeTypeFactory()
    {
      while (!TreeTypes_.empty())
      {
        map<char, TreeType*>::iterator it = TreeTypes_.begin();
        delete it->second;
        TreeTypes_.erase(it);
      }
    }
    TreeType* GetTreeType(char key)
    {
      TreeType* TreeType = NULL;
      if (TreeTypes_.find(key) != TreeTypes_.end())
      {
        TreeType = TreeTypes_[key];
      }
      else
```

```
    {
      switch (key)
      {
      case 'A':
        TreeType = new TreeTypeA();
        break;
      case 'B':
        TreeType = new TreeTypeB();
        break;
        //…
      case 'Z':
        TreeType = new TreeTypeZ();
        break;
      default:
        cout << "Not Implemented" << endl;
        throw("Not Implemented");
      }
      TreeTypes_[key] = TreeType;
    }
    return TreeType;
  }
private:
  map<char, TreeType*> TreeTypes_;
};


//The Main method
int main()
{
  string forestType = "ZAZZBAZZBZZAZZ";
  const char* chars = forestType.c_str();

  TreeTypeFactory* factory = new TreeTypeFactory;

  // extrinsic state
  int size = 10;

  // For each TreeType use a flyweight object
  for (size_t i = 0; i < forestType.length(); i++)
  {
    size++;
    TreeType* TreeType = factory->GetTreeType(chars[i]);
```

```
        TreeType->Display(size);
    }

    //Clean memory
    delete factory;

    _getch();
    return 0;
}
```

How it works...

In this example, we have created a forest. The basic principle of the flyweight pattern is applied, whereby part of the structure is shared across all trees and part is dictated by the client. In this example, apart from the size (this could be anything, size is just chosen to be an example), every other attribute is chosen to be shared. We create a tree-type interface which contains all the attributes. We then have derived classes that have their attributes overridden and a method to set the `size` attribute. We can have multiple such trees. Generally, the greater the variety of trees, the more detailed the forest will look. Let us say that we have 10 different types of tree, so we need to have 10 different classes that derive from the interface and have a method to assign the `size` attribute from the client size.

Finally, we have the tree factory, which assigns each tree at runtime. We create a reference to the interface as we do with any factory pattern. However, instead of directly instantiating a new object, we first check the map to see whether the tree's attributes are already present. If they are not, we assign a new object and push the attributes to the map. So the next time a request comes for a similar tree structure to one that has already been assigned, we can share the attributes from the map. Finally, from the client, we create a forest-type document which we feed to the factory, and it generates the forest for us using the trees listed in the document. As the majority of the attributes are shared, the memory footprint is very low. The following screenshot shows us how the forest is created:

```
C:\Users\Druhin\Desktop\Book\C++Book\C++forGameDevelopment\Book\Chapter6\Sou...
Specify the different building types in your village
Barracks(1) Temple(2) Farmhouse(3) Go(0): 1
Barracks(1) Temple(2) Farmhouse(3) Go(0): 2
Barracks(1) Temple(2) Farmhouse(3) Go(0): 3
Barracks(1) Temple(2) Farmhouse(3) Go(0): 2
Barracks(1) Temple(2) Farmhouse(3) Go(0): 3
Barracks(1) Temple(2) Farmhouse(3) Go(0): 1
Barracks(1) Temple(2) Farmhouse(3) Go(0): 0

There are total 6 buildings
Health of Barrack is :100
Health of Temple is :75
Health of Farmhouse is :50
Health of Temple is :75
Health of Farmhouse is :50
Health of Barrack is :100
```

Using the strategy pattern

The strategy design pattern is a very smart way of designing code. In games, this is mostly used for the AI component. In this pattern, we define a large number of algorithms and have all of them from a common interface signature. Then at runtime, we can change the clients of the algorithms. So in effect, the algorithms are independent of the clients.

Getting ready

To work through this recipe, you will need a machine running Windows. You also need to have a working copy of Visual Studio installed on your Windows machine. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to implement the strategy pattern:

1. Open Visual Studio.
2. Create a new C++ project.
3. Select a Win32 console application.
4. Add a `Source.cpp` file.

5. Add the following lines of code to it:

```
#include <iostream>
#include <conio.h>

using namespace std;


class SpecialPower
{
public:
  virtual void power() = 0;
};

class Fire : public SpecialPower
{
public:
  void power()
  {
    cout << "My power is fire" << endl;
  }
};

class Invisibility : public SpecialPower
{
public:
  void power()
  {
    cout << "My power is invisibility" << endl;
  }
};




class FlyBehaviour
{
public:
  virtual void fly() = 0;
};

class FlyWithWings : public FlyBehaviour
{
public:
  void fly()
```

```
  {
    cout << "I can fly" << endl;
  }
};

class FlyNoWay : public FlyBehaviour
{
public:
  void fly()
  {
    cout << "I can't fly!" << endl;
  }
};

class FlyWithRocket : public FlyBehaviour
{
public:
  void fly()
  {
    cout << "I have a jetpack" << endl;
  }
};




class Enemy
{

public:

  SpecialPower *specialPower;
  FlyBehaviour   *flyBehaviour;

  void performPower()
  {
    specialPower->power();
  }

  void setSpecialPower(SpecialPower *qb)
  {
    cout << "Changing special power..." << endl;
```

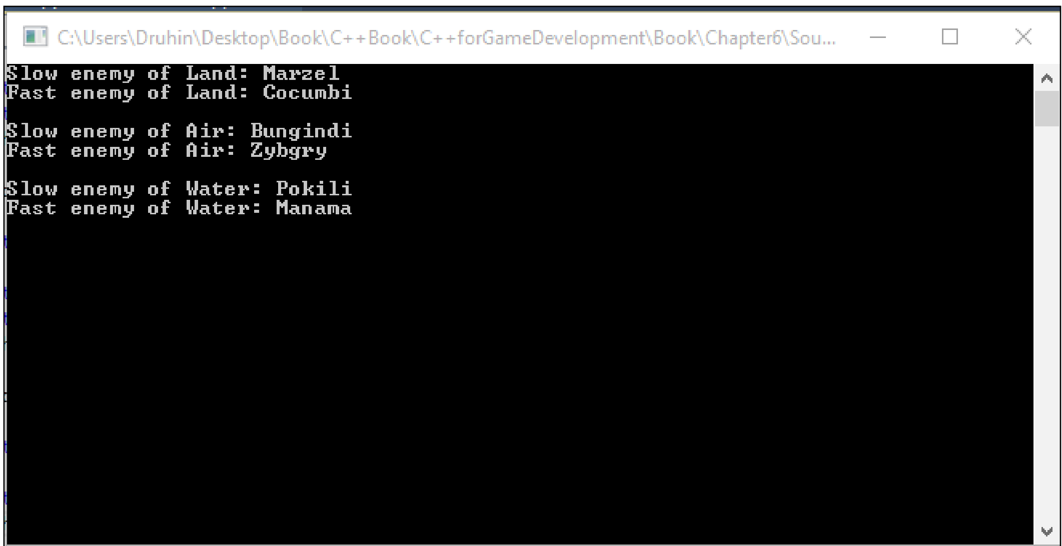
```
    specialPower = qb;
  }

  void performFly()
  {
    flyBehaviour->fly();
  }

  void setFlyBehaviour(FlyBehaviour *fb)
  {
    cout << "Changing fly behaviour..." << endl;
    flyBehaviour = fb;
  }

  void floatAround()
  {
    cout << "I can float." << endl;
  }

  virtual void display() = 0; // Make this an abstract class by
having a pure virtual function


};

class Dragon : public Enemy
{
public:
  Dragon()
  {
    specialPower = new Fire();
    flyBehaviour = new FlyWithWings();
  }

  void display()
  {
    cout << "I'm a dragon" << endl;
  }

};

class Soldier : public Enemy
{
public:
```

```
    Soldier()
    {
      specialPower = new Invisibility();
      flyBehaviour = new FlyNoWay();
    }

    void display()
    {
      cout << "I'm a soldier" << endl;
    }
  };




  int main()
  {
    Enemy *dragon = new Dragon();
    dragon->display();
    dragon->floatAround();
    dragon->performFly();
    dragon->performPower();

    cout << endl << endl;

    Enemy *soldier = new Soldier();
    soldier->display();
    soldier->floatAround();
    soldier->performFly();
    soldier->setFlyBehaviour(new FlyWithRocket);
    soldier->performFly();
    soldier->performPower();
    soldier->setSpecialPower(new Fire);
    soldier->performPower();

    _getch();
    return 0;
  }
```

How it works...

In this example, we have created different interfaces for different properties that the enemy may have. So, since we know that special power is a property every enemy type will have, we have created an interface called `SpecialPower` and then derived two classes from it called `Fire` and `Invisibility`. We can add as many special powers as we want, we just need to create a new class and derive from the special power interface. Similarly, all the enemy types should have a property for flying. Either they fly, or don't fly, or fly with the help of jetpacks.

So we have created a `FlyBehaviour` interface and have the different flying-type classes derive from it. After that, we have created an abstract class for the enemy type which contains both the interfaces as references. Hence any derived class can decide what flying type and what special power it needs. This also gives us the flexibility to change special powers and flying ability at runtime. The screenshot below shows a brief example of this design pattern:



Using the command design pattern

The command design pattern generally involves encapsulating a command as an object. This is highly used in networking for games, in which player movements are sent across as objects that are run as commands. The four main points to remember in a command design pattern are the client, invoker, receiver, and command. The command object has knowledge of the receiver object. The receiver does the work after it receives a command. The invoker performs the command, without having any knowledge of who has sent the command. The client controls the invoker and decides which commands are to be performed at which stage.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to implement the command pattern:

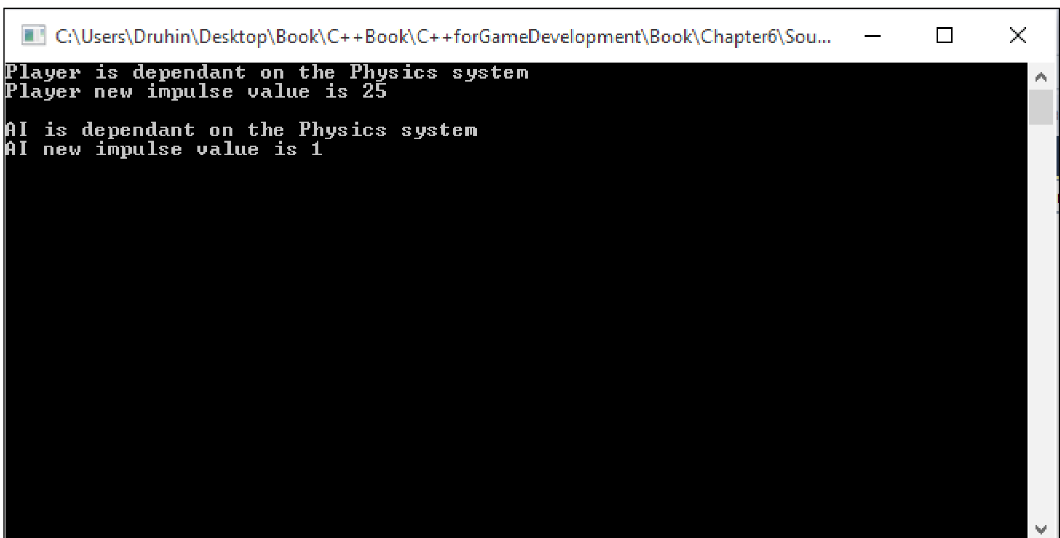
1. Open Visual Studio.
2. Create a new C++ project console application.
3. Add the following lines of code:

```
#include <iostream>
#include <conio.h>

using namespace std;
class NetworkProtocolCommand
{
public:
  virtual void PerformAction() = 0;
};
class ServerReceiver
{
public:
  void Action()
  {
    cout << "Network Protocol Command received" <<endl;

  }
};
class ClientInvoker
{
  NetworkProtocolCommand *m_NetworkProtocolCommand;

public:
  ClientInvoker(NetworkProtocolCommand *cmd = 0) : m_
NetworkProtocolCommand(cmd)
  {
  }

  void SetCommad(NetworkProtocolCommand *cmd)
  {
    m_NetworkProtocolCommand = cmd;
```

```
    }

    void Invoke()
    {
      if (0 != m_NetworkProtocolCommand)
      {
        m_NetworkProtocolCommand->PerformAction();
      }
    }
};

class MyNetworkProtocolCommand : public NetworkProtocolCommand
{
    ServerReceiver *m_ServerReceiver;

public:
    MyNetworkProtocolCommand(ServerReceiver *rcv = 0) : m_
ServerReceiver(rcv)
    {
    }

    void SetServerReceiver(ServerReceiver *rcv)
    {
      m_ServerReceiver = rcv;
    }

    virtual void PerformAction()
    {
      if (0 != m_ServerReceiver)
      {
        m_ServerReceiver->Action();
      }
    }
};

int main()
{
    ServerReceiver r;
    MyNetworkProtocolCommand cmd(&r);
    ClientInvoker caller(&cmd);
    caller.Invoke();

    _getch();
    return 0;
}
```

How it works...

As we can see in this example, we have set up an interface to send information via the network protocol command. From that interface, we can derive multiple child instances to be used on the client side. We then need to create a server receiver, which will receive commands sent from the client. We also need to create a client invoker, which will invoke the command. A reference to the network protocol command should also be present in this class. Finally, from the client side, we need to create an instance of the server and attach the instance to the object of the network protocol command's child that we created. We then use the client invoker to invoke the command and send it via the network protocol command to the receiver. This ensures that an abstraction is maintained and the entire message is send via packets. The following screenshot shows a part of the process:



Creating an advanced game using design patterns

After understanding the basic design patterns, it's important to combine them to create a good game. It takes years of practice to finally understand what architecture will suit the game structure. More often than not, we have to use a few design patterns in conjunction to come up with clean code that can be applied to the game. The factory pattern will probably be your most used design pattern, but that is purely an anecdotal reference from my experience.

Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

How to do it...

In this recipe we will find out how easy it is to combine design patterns to create a game:

1. Open Visual Studio.
2. Create a new C++ project console application.
3. Add the following lines of code:

```
#ifndef _ISPEED_H
#define _SPEED_H

class ISpeed
{
  public:
    virtual void speed() = 0;

};

#end
#ifndef _ISPECIALPOWER
#define _ISPECIALPOWER
class ISpecialPower
{
public:
  virtual void power() = 0;
};
#endif


#ifndef _IENEMY_H
#define _IENEMY_H

#include "ISpecialPower.h"
#include "ISpeed.h"



class IEnemy
{

public:

  ISpecialPower *specialPower;
```

```
  ISpeed   *speed;

  void performPower()
  {
    specialPower->power();
  }

  void setSpecialPower(ISpecialPower *qb)
  {

  }


};
#endif
#include <iostream>
#include "ISpeed.h"

#pragma once
class HighSpeed :public ISpeed
{
public:
  HighSpeed();
  ~HighSpeed();
};

#include "IEnemy.h"

class Invisibility;
class HighSpeed;

class Soldier : public IEnemy
{
public:
  Soldier()
  {

  }


};
```

How it works...

The previous code is just a small section of the code. Let us assume that we need to make a game where there are different classes of enemies and there are different types of powers, as well as some special boosts or power-ups. One approach to this is to think of all the powers and special boosts as individual classes that derive from an interface. So we need to create an interface for speed, which can be derived from the `HighSpeed` class and so on. Similarly, we can create a `SpecialPower` interface, which can be derived by the `Fire` class and so on. We need to create interfaces for all the groups of properties that our characters can have. Finally, we need to create an interface for the characters (`IEnemy`) that is derived by the `Soldier`, `Archer`, and `Grenadier` class, and so on. The `IEnemy` interface should also hold a reference to all the other interfaces, such as `ISpecialPower` and `ISpeed`. In this way, the child classes of `IEnemy` can decide what power and speed they want to have. This is similar to the strategy design pattern. We can further improve this structure if we want to group the enemies into types, let's say land enemies and air enemies. In that case, either we create an interface for `IType` and have `Land` and `Air` classes derive from it, or we could create a factory that creates enemy types for us depending on the type the client requests. Each enemy type created will also be a derived class from `IEnemy`, so that it will also have the references to the previous interfaces. As the complexity of the game increases, we can add more design patterns to aid us.

7

Organizing and Backing Up

In this chapter, the following recipes will be covered:

- Versions of source control
- Installing a versioning client
- Selecting a host to save your data
- Adding source control to your code – committing and updating your code
- Resolving conflicts
- Creating a branch

Introduction

Let us consider that we need to work on a project that has many developers. If every developer is working on different source files, one (rather horrible) way to work is to get the newly updated source file in an e-mail or an FTP client and replace it in your project. Now what if the developers, including yourself, are working on the same source file. We can still follow this horrible way and add the parts that we have worked on to the file we received via FTP, but very soon this is going to become very cumbersome and make it almost impossible to work. So we have a system of saving the files to some central repository or distributed repository, and then have the means to update and send the code so that every developer is working with the latest copy. There are various ways to perform this and it is commonly referred to as versioning the code.

Versions of source control

Revision control is a very effective way to share files across developers. There are various version control systems and each has its own merits and drawbacks. We will be looking at the three most popular version control systems out there.

Getting ready

To work through this recipe, you will need a machine running Windows. No other prerequisites are required.

How to do it...

In this recipe, we will see the different types of source control available to us:

1. Go to this link and download for an SVN client: `http://tortoisesvn.net/downloads.html`
2. Go to this link and download for a GIT client: `https://desktop.github.com`
3. Go to this link and download for a Mercurial client: `http://tortoisehg.bitbucket.org/download/index.html`

How it works...

There are various types of SVN clients available to us. Each has its own merits and drawbacks.

SVN has a lot of features that fix issues relating to atomic operations and source corruption. It is free and open source. It has lots of plugins for different IDEs. However, one of the major drawbacks of this tool is that is comparatively very slow in its operations.

GIT was made primarily for Linux but it improves the operation speed a lot. It does work on UNIX systems as well. It has cheap branch operations but it is not totally optimized for a single developer and its Windows support is limited compared to Linux. However, GIT is extremely popular and many prefer GIT over SVN.

Installing a versioning client

There are plenty of versioning clients. However, we are going to look at an SVN client. **Tortoise SVN** is by far the most popular among SVN users. Although GIT is another system that is immensely popular, we will look at Tortoise SVN for this recipe. Tortoise SVN provides a very friendly and intuitive interface, so it is very easy for beginners to grasp as well. Within a few hours, a total newbie can understand the basics of using Tortoise SVN.

Getting ready

You need a Windows machine. No other prerequisite is needed.

How to do it...

In this recipe, we will find out how easy it is to install and use Tortoise SVN:

1. Go to this link: `http://tortoisesvn.net/downloads.html`
2. Download and install the correct version, based on whether you are using a 32-bit or a 64-bit Windows machine.
3. Create a new folder on your computer.
4. Right-click on the folder.
5. Check that a new command called **SVN Checkout...** is now available for use.

How it works...

After we go the download site and install the package, it gets installed on the system and also lots of shell and kernel commands are added. So when we right-click on the folder, the **SVN Checkout...** command is now added as a property for any new folder. There is also another command called **Tortoise SVN** available to us, which has even more commands. After we check out a project, the **SVN Checkout...** gets replaced with **SVN Update** or **SVN Commit**. We just need to make sure that we have added the correct installer to the machine, based on the OS version we are using.

Selecting a host to save your data

Before we can start versioning our code, we need to decide where we need to save our code files to. There are quite a few ways to do this, but we will discuss the two most popular ways. The first way is to save the files locally and treat your personal computer as a server to host data. The second method is to use a cloud service to host the data files for us.

Getting ready

You need to have a working Windows machine.

How to do it...

In this recipe, we will find out how easy it is to host the files locally or on the cloud.

For the files saved on the cloud follow these steps:

1. Go to the following link: `https://xp-dev.com`.
2. Go to **Plans** and select a plan most suitable to your needs. There is also a free plan for 10 MB.
3. After selecting a plan, you will be redirected to create a name for the current project.
4. The new project will now show up on the dashboard. You can create multiple projects based on your plan.
5. Click on a project. This should open up more tabs. The most important ones currently are:
    - **Repository**
    - **Project Tracking**
    - **Activity**
    - **Settings**
6. Click on **Repository** to create a new repository.
7. The link generated can now be used to version our files in the project.
8. To add users to the project, click on **Settings** and invite users to the project.

For the files saved on the local server:

1. Save the new project or an empty project on your computer.
2. Download **Visual SVN Server** from here: `https://www.visualsvn.com/server/`.
3. Install the software.
4. Then create a project from the existing project.
5. Your project is now ready to be version controlled.
6. To add users, click on **Users** and add a username and password.

How it works...

When we create a project on `xp-dev`, what actually happens is that `xp-dev` creates a cloud space for us on their server, based on whatever plan we have chosen. After that, for each iteration of the file, it saves a copy on the server. On the dashboard, once we create one repository, we can create a new repository and the URL generated will now be the URL of the project. In that way, we can revert back to any iteration or restore a file if we mistakenly delete it. When we commit a file, a new copy of the file is now saved on the server. When we update the project, the latest version on the server is now pushed to your local machine. In this way, `xp-dev` saves the entire history of activities for all updates and commits. The drawback of the system is that if the `xp-dev` client is down, then all the cloud services will also be down. Hence, the project will suffer due to you not being able to do any updates or commits.

The other way to host is to use your own local machine. Visual SVN Server basically turns your computer into a server. After that, the process is pretty similar to how `xp-dev` handles all updates and commits.

What we could also do is take some space from Amazon or Azure and use that space as a server. In that case, the steps are pretty similar to the second method (local server). After logging in to Amazon space or Azure space, treat that as your machine and then repeat the steps for the local server.

Adding source control – committing and updating your code

One of the most important things that you can do to files when working on a collaborative project or individually is to add source control. The biggest advantage of doing so is that the files are always backed up and versioned. Let's say that you made some local changes and there are lots of crashes. As a result of those crashes, what will you do? One option is to retrace your steps and change them back to what they were before. This is a time-wasting process and there is also risk involved. If your files are backed up, all you need to do is a revert operation to a particular revision and the code is restored to that point. Similarly, if we delete a file by mistake, we can always update the project and it will pull the most current file from the server.

Getting ready

For this recipe, you will need a Windows machine and an installed version of an SVN client. A data hosting service should already be integrated by now and you should have a URL. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to add source control:

1. Create a new folder on the machine.
2. Rename it to whatever you want to call it.
3. Right-click and check whether the SVN command is showing up as one of the options.
4. Click on **SVN Checkout**. Use the URL you received from `xp-dev` or your local server or cloud server.
5. Add a file into the new folder. It can be in any format.
6. Right-click on the file and select **Tortoise SVN |Add**.
7. Go to the root folder and select **SVN** | **Commit**.
8. Delete the file.
9. Go to **SVN** | **Update**.
10. Make some changes to the file.
11. Select **SVN** | **Commit**.
12. Then select **Tortoise SVN** and then **Revert to this revision** (revision `1`).

How it works...

After the SVN checkout is successful, the project is either copied from the local machine to the server or copied from the server to the local machine, based on which is the most up to date. Once we add the file into the folder, we have to remember that the file is still local. Only we can see it and have access to it. Others who are working on that project will have no idea about it. Now, one of the common mistakes that a new programmer may make at this stage is to forget to add the file to the SVN. When you commit the project, that file will not show up. There is a checkbox in the commit section for **Show unversioned files**. However, I will not recommend that approach, as all temporary files will also be shown in this case. A better approach is to right-click on the file and go to **Tortoise SVN** | **Add**. This will add the file for revisioning. Now we can do an SVN commit and the file will be stored on the server.

When we delete the file, we again have to remember that we have just deleted the file locally. The instance of it still exists on the server. So when we perform an SVN update, the file will again be restored. We have to be careful not to do a **Tortoise SVN** | **Delete and Commit**. This will remove it from the server for that revision. Now if we make some changes to the file, we can **SVN Commit** it. We no longer need to select **Tortoise SVN** | **Add**. This creates a new version of the file on the server. Both versions of the file are now present. We can have as many versions as we need. To access any revision, we need to select either the root folder or any particular file and perform a **Revert to this revision** `(number)`. The server then looks up the version that we requested and pushes the correct copy to us.

Resolving conflicts

Let us consider a single source file that has been worked on by multiple programmers. You might have some local changes. When you try to update, it may happen that the SVN client is smart enough to merge the files together. However, in most cases it will not be able to merge properly and we need to resolve conflicts effectively. The SVN client, however, will show the files that are in conflict.

Getting ready

For this recipe, you will need a Windows machine and an installed version of an SVN client. A versioned project is also necessary.

How to do it...

In this recipe, we will find out how easy it is to resolve conflicts:

1. Take a project that is already versioned and committed to SVN.
2. Open a file in an editor and make changes to the file.
3. Perform the **SVN Update** operation.
4. The files now show a conflict.
5. See the differences between the two files using the **Diff tool** or **Win Merge** (you may need to install Win Merge separately).
6. Generally, the left-hand side will be the local revision and the right-hand side will be the version on the server. However, these could be swapped as well.
7. After looking at the differences, you may resolve the conflicts in two ways:
    - Select the portions that you want from the server and the portions that you want from the local changes.
    - Select **Resolve conflict using "mine"** or select **Resolve conflict using "theirs"**.

How it works...

What happens in a conflict is that the client on its own cannot make a decision on whether the local copy or the server copy should be treated as the correct working version. Most good clients will show this as an error once we do an update. Other clients will insert both sections in the code, generally with an `r>>>>>>` or an `m>>>>` notation, to denote which section is the server and which section is ours. On the Tortoise SVN, if we choose to ignore conflicts, then these notations may be displayed as separate files or included in the file. A better approach is to always resolve conflicts. If we use a tool such as Win Merge, it will show us the two revisions side by side and we can compare and choose the sections we need, or the whole file. After that, once we have taken the changes and committed them, that file will become the updated version on the server. So others updating their code will also get the changes we made.

Creating a branch

Let us consider that we are making a game which is due for release at the end of the year. However, we also need to showcase a polished version of the game for GDC or E3. At that point, the producer might ask us to make a build specific to E3 or GDC. This GDC or E3 build can be polished and made stable, whereas the main build may continue to be experimented with by adding new features.

Getting ready

To work through this recipe, you will need a machine running Windows with an installed version of an SVN client. A versioned project is also required. No other prerequisites are needed.

How to do it...

In this recipe, we will find out how easy it is to create a branch:

1. Right-click on the versioned project.
2. Go to the repo browser.
3. Select the root folder from which you want to create the branch.
4. Select the destination.
5. A branch is now created.
6. Check out the created branch onto the machine by using the URL.

How it works...

When we create a branch from a root folder, a mirror copy of that folder and consequent subfolders is created. From then on, the two can work independently. The main root has a URL, and the branch also has its own URL. We can update and commit to the branch as we would for the root folder. Also, all other functionalities are available for the branch as usual. Sometimes, after we make changes to the branch, we might need to push them back to the root. Although the SVN client, Tortoise SVN, provides us with a tool to merge the branches, it is rarely successful and more often than not we need to do the merge manually.

8
AI in Game Development

In this chapter, the following recipes will be covered:

- Adding artificial intelligence to a game
- Using heuristics in a game
- Using a Binary Space Partition Tree
- Creating a decision making AI
- Adding behavioral movements
- Using neural network
- Using genetic algorithms
- Using other waypoint systems

Introduction

Artificial intelligence (**AI**) can be defined in many ways. Artificial intelligence deals with finding similarities in different situations and differences in similar situations. AI can help to bring realism to a game. The user playing the game should feel that that entity that they are competing against is another human. Achieving this is extremely difficult and can consume a lot of processing cycles. In fact, there is a *turing test* held every year to determine whether an AI can fool other humans into believing that it is human. Now, if we use a lot of processing cycles for the AI, then executing the game at above 40 FPS can become extremely difficult. Hence we need to write efficient algorithms to achieve this.

Adding artificial intelligence to a game

Adding artificial intelligence to a game may be easy or extremely difficult, based on the level of realism or complexity we are trying to achieve. In this recipe, we will start with the basics of adding artificial intelligence.

Getting ready

To work through this recipe, you will need a machine running Windows and a version of Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to add a basic artificial intelligence to the game. Add a source file called `Source.cpp`. Add the following code to it:

```
// Basic AI : Keyword identification

#include <iostream>
#include <string>
#include <string.h>


std::string arr[] = { "Hello, what is your name ?", "My name is Siri"
};

int main()
{

  std::string UserResponse;

  std::cout << "Enter your question? ";
  std::cin >> UserResponse;

  if (UserResponse == "Hi")
  {
    std::cout << arr[0] << std::endl;
    std::cout << arr[1];
  }


  int a;
  std::cin >> a;
  return 0;

}
```

How it works...

In the previous example, we are using a string array to store a response. The idea of the software is to create an intelligent chat bot that can reply to questions asked by users and interact with them as if it were human. Hence the first task was to create an array of responses. The next thing to do is to ask the user for the question. In this example, we are searching for a basic keyword called `Hi` and, based on that, we are displaying the appropriate answer. Of course, this is a very basic implementation. Ideally we would have a list of keywords and responses when either of the keywords is triggered. We can even personalize this by asking the user for their name and then appending it to the answer every time.

The user may also ask to search for something. That is actually quite an easy thing to do. If we have detected the word that the user is longing to search for correctly, we just need to enter that into the search engine. Whatever result the page displays, we can report it back to the user. We can also use voice commands to enter the questions and give the responses. In this case, we would also need to implement some kind of **NLP** (**Natural Language Processing**). After the voice command is correctly identified, all the other processes are exactly the same.

Using heuristics in a game

Adding heuristics in a game means to define rules. We need to define a set of rules for the AI agent so that it can move to its destination in the best possible way. For example, if we want to write a pathfinding algorithm, and define only its start and end positions, it may get there in many different ways. However, if we want the agent to reach the goal in a particular way, we need to establish a heuristic function for it.

Getting ready

You need a Windows machine and a working copy of Visual Studio. No other prerequisites are required.

How to do it...

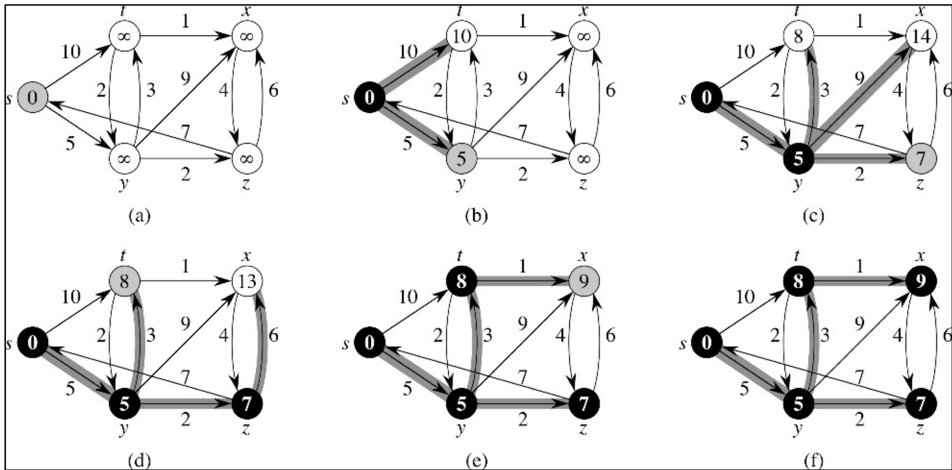
In this recipe, we will find out how easy it is to add a heuristic function to our game for pathfinding. Add a source file called `Source.cpp` and add the following code to it:

```
for (auto next : graph.neighbors(current)) {
  int new_cost = cost_so_far[current] + graph.cost(current, next);
  if (!cost_so_far.count(next) || new_cost < cost_so_far[next]) {
    cost_so_far[next] = new_cost;
    int priority = new_cost + heuristic(next, goal);
    frontier.put(next, priority);
    came_from[next] = current;
  }
```

How it works...

There are many ways to define what a heuristic is. However, the simplest way to think about it is that it is a function that provides hints and directions for the AI to reach a specified goal. Let us say that our AI needs to go from point `A` to point `D`. Now, there are also points `B` and `C` somewhere on the map. How should the AI decide which path to take? This is what is provided by a heuristic function. In this example, we have used a heuristic in a pathfinding algorithm called `A*`. In special cases where the heuristic function is `0`, we get an algorithm called **Dijkstra's**.

Let us consider Dijkstra's first. It will be easier to understand `A*` later.



Let us consider we need to find the shortest path between **s** and **x**, traversing all nodes at least once. **s**, **t**, **y**, **x**, and **z** are the different nodes or the different subdestinations. The number from one node to another node is the cost of going from one node to the other. The algorithm states that we start from **s** with a **0** value and consider all other nodes to be infinite. The next thing to consider is the nodes adjacent to **s**. The nodes adjacent to **s** are **t** and **y**. The cost of reaching them is **5** and **10** respectively. We note that and then replace the infinity value at those nodes with **5** and **10**. Now let us consider the node **y**. The adjacent nodes are **t**, **x**, and **z**. The cost to reach **x** is **5** (its current node value) plus **9** (path cost value) equals *14*. Similarly, the cost to reach **z** is *5 + 2 = 7*. So we replace the infinity values of **x** and **z** with **14** and **7** respectively. Now, the cost to reach **t** is *5 + 3 = 8*. However, it already has a node value. Its value is **10**. Since *8<10*, we will replace **t** with **8**. We keep on doing this for all the nodes. After that we will get the minimum cost to traverse all the nodes.

`A*` has two cost functions:

- `g(x)`: This is the same as Dijkstra. It is the real cost to reach node **x**.
- `h(x)`: This is the approximate cost from node **x** to the goal node. It is a heuristic function. This heuristic function should never overestimate the cost. That means the real cost to reach goal node from node **x** should be greater than or equal to `h(x)`. It is called an admissible heuristic.

The total cost of each node is calculated using *f(x) = g(x)+h(x)*.

In `A*`, we do not need to traverse all nodes, we just need to find the minimum path from start to the destination. An A* search only expands a node if it seems promising. It only focuses on reaching the goal node from the current node, not reaching every other node. It is optimal if the heuristic function is admissible. So writing the heuristic function is the key to checking whether to expand to a node or not. In the previous example, we used neighboring nodes and formed a priority list to decide that.

Using a Binary Space Partition Tree

Sometimes in games we work with a lot of geometry and huge 3D worlds. If our game camera was to render all of it all the time, then it would be extremely expensive and the game would not be able to run smoothly at higher frame rates. Hence we need to write intelligent algorithms so that the world is divided into more manageable chunks that can be traversed easily using a tree structure.

Getting ready

You need to have a working Windows machine and a working copy of Visual Studio.

How to do it...

Add a source file called `Source.cpp`. Then add the following code to it:

```
sNode(elemVec& toProcess, const T_treeAdaptor& adap)
      : m_pFront(NULL)
      , m_pBack(NULL)
    {
      // Setup
      elemVec frontVec, backVec;
      frontVec.reserve(toProcess.size());
      backVec.reserve(toProcess.size());

      // Choose which node we're going to use.
```

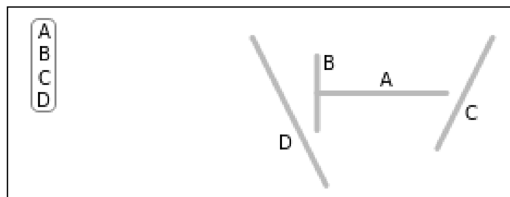
```
    adap.ChooseHyperplane(toProcess, &m_hp);

    // Iterate across the rest of the polygons
    elemVec::iterator iter = toProcess.begin();
    for (; iter != toProcess.end(); ++iter)
    {
      T_element front, back;
      switch (adap.Classify(m_hp, *iter))
      {
      case BSP_RELAT_IN_FRONT:
        frontVec.push_back(*iter);
        break;
     <...>
    }

    // Now recurse if necessary
    if (!frontVec.empty())
      m_pFront = new sNode(frontVec, adap);
    if (!backVec.empty())
      m_pBack = new sNode(backVec, adap);
  }

  sNode(std::istream& in)
  {
    // First char is the child state
    // (0x1 means front child, 0x2 means back child)
    int childState;
    in >> childState;

    // Next is the hyperplane for the node
    in >> m_hp;

    // Next is the number of elements in the node
    unsigned int nElem;
    in >> nElem;
    m_contents.reserve(nElem);

    while (nElem--)
    {
      T_element elem;
      in >> elem;
```

```
        m_contents.push_back(elem);
      }

      // recurse if we have children.
      if (childState & 0x1)
        m_pFront = new sNode(in);
      else
        m_pFront = NULL;
      if (childState & 0x2)
        m_pBack = new sNode(in);
      else
        m_pBack = NULL;
    }
```

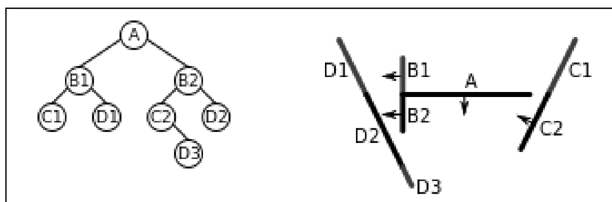
How it works...

A **Binary Space Partition** (**BSP**) tree, as the name implies, is a tree structure within which a geometrical space is partitioned. To be more precise, in BSP a plane is portioned into more hyperplanes. A plane is such that it has one dimension less than the ambient space from which it was created. So a 3D plane would have 2D hyperplanes and a 2D plane would have 1D lines. The idea behind this is once we have divided the planes into these hyperplanes in a logical manner, we can save the formation into a tree structure. Finally, we can traverse the tree structure in real time to provide better frame rates for the game overall.

Let us consider an example in which the world looks like the following diagram. The camera must decide which areas it should render and which it should not. Hence, dividing them using a logical algorithm is necessary:



After we apply the algorithm, the tree structure should look like the following:



Finally, we traverse this algorithm as with any other tree structure, using the concept of parent and child, and we get the desired sections that the camera should render.

Creating a decision making AI

A **decision tree** is one of the most useful things to have in machine learning for AI. Given a large number of scenarios, based on certain parameters, decision making is essential. If we can write a system that can make these decisions well, then we can not only have a well-written algorithm but also have a lot of unpredictability in terms of gameplay. This will add a lot of variation to the game and will help the *replayability* of the overall game.

Getting ready

For this recipe, you will need a Windows machine and Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to add source control:

```
/* Decision Making AI*/


#include <iostream>
#include <ctime>


using namespace std;

class TreeNodes
{
public:
  //tree node functions
  TreeNodes(int nodeID/*, string QA*/);
  TreeNodes();

  virtual ~TreeNodes();

  int m_NodeID;

  TreeNodes* PrimaryBranch;
```

```
    TreeNodes* SecondaryBranch;
  };


  //constructor
  TreeNodes::TreeNodes()
  {
    PrimaryBranch = NULL;
    SecondaryBranch = NULL;

    m_NodeID = 0;
  }

  //deconstructor
  TreeNodes::~TreeNodes()
  { }

  //Step 3! Also step 7 hah!
  TreeNodes::TreeNodes(int nodeID/*, string NQA*/)
  {
    //create tree node with a specific node ID
    m_NodeID = nodeID;

    //reset nodes/make sure! that they are null. I wont have any funny
  business #s -_-
    PrimaryBranch = NULL;
    SecondaryBranch = NULL;
  }

  //the decision tree class
  class DecisionTree
  {
  public:
    //functions
    void RemoveNode(TreeNodes* node);
    void DisplayTree(TreeNodes* CurrentNode);
    void Output();
    void Query();
    void QueryTree(TreeNodes* rootNode);
    void PrimaryNode(int ExistingNodeID, int NewNodeID);
    void SecondaryNode(int ExistingNodeID, int NewNodeID);
    void CreateRootNode(int NodeID);
```

```
    void MakeDecision(TreeNodes* node);

    bool SearchPrimaryNode(TreeNodes* CurrentNode, int ExistingNodeID,
  int NewNodeID);
    bool SearchSecondaryNode(TreeNodes* CurrentNode, int ExistingNodeID,
  int NewNodeID);

    TreeNodes* m_RootNode;

    DecisionTree();

    virtual ~DecisionTree();
  };

  int random(int upperLimit);

  //for random variables that will effect decisions/node values/weights
  int random(int upperLimit)
  {
    int randNum = rand() % upperLimit;
    return randNum;
  }

  //constructor
  //Step 1!
  DecisionTree::DecisionTree()
  {
    //set root node to null on tree creation
    //beginning of tree creation
    m_RootNode = NULL;
  }

  //destructor
  //Final Step in a sense
  DecisionTree::~DecisionTree()
  {
    RemoveNode(m_RootNode);
  }

  //Step 2!
  void DecisionTree::CreateRootNode(int NodeID)
  {
    //create root node with specific ID
```

```
    // In MO, you may want to use thestatic creation of IDs like with
entities. depends on how many nodes you plan to have
    //or have instantaneously created nodes/changing nodes
    m_RootNode = new TreeNodes(NodeID);
}

//Step 5.1!~
void DecisionTree::PrimaryNode(int ExistingNodeID, int NewNodeID)
{
    //check to make sure you have a root node. can't add another node
without a root node
    if (m_RootNode == NULL)
    {
      cout << "ERROR - No Root Node";
      return;
    }

    if (SearchPrimaryNode(m_RootNode, ExistingNodeID, NewNodeID))
    {
      cout << "Added Node Type1 With ID " << NewNodeID << " onto Branch
Level " << ExistingNodeID << endl;
    }
    else
    {
      //check
      cout << "Node: " << ExistingNodeID << " Not Found.";
    }
}

//Step 6.1!~ search and add new node to current node
bool DecisionTree::SearchPrimaryNode(TreeNodes *CurrentNode, int
ExistingNodeID, int NewNodeID)
{
    //if there is a node
    if (CurrentNode->m_NodeID == ExistingNodeID)
    {
      //create the node
      if (CurrentNode->PrimaryBranch == NULL)
      {
        CurrentNode->PrimaryBranch = new TreeNodes(NewNodeID);
      }
      else
      {
        CurrentNode->PrimaryBranch = new TreeNodes(NewNodeID);
      }
```

```
      return true;
    }
    else
    {
      //try branch if it exists
      //for a third, add another one of these too!
      if (CurrentNode->PrimaryBranch != NULL)
      {
        if (SearchPrimaryNode(CurrentNode->PrimaryBranch,
ExistingNodeID, NewNodeID))
        {
          return true;
        }
        else
        {
          //try second branch if it exists
          if (CurrentNode->SecondaryBranch != NULL)
          {
            return(SearchSecondaryNode(CurrentNode->SecondaryBranch,
ExistingNodeID, NewNodeID));
          }
          else
          {
            return false;
          }
        }
      }
      return false;
    }
}

//Step 5.2!~    does same thing as node 1.  if you wanted to have more
decisions,
//create a node 3 which would be the same as this maybe with small
differences
void DecisionTree::SecondaryNode(int ExistingNodeID, int NewNodeID)
{
  if (m_RootNode == NULL)
  {
    cout << "ERROR - No Root Node";
  }

  if (SearchSecondaryNode(m_RootNode, ExistingNodeID, NewNodeID))
  {
```

```
    cout << "Added Node Type2 With ID " << NewNodeID << " onto Branch
Level " << ExistingNodeID << endl;
  }
  else
  {
    cout << "Node: " << ExistingNodeID << " Not Found.";
  }
}

//Step 6.2!~ search and add new node to current node
//as stated earlier, make one for 3rd node if there was meant to be
one
bool DecisionTree::SearchSecondaryNode(TreeNodes *CurrentNode, int
ExistingNodeID, int NewNodeID)
{
  if (CurrentNode->m_NodeID == ExistingNodeID)
  {
    //create the node
    if (CurrentNode->SecondaryBranch == NULL)
    {
      CurrentNode->SecondaryBranch = new TreeNodes(NewNodeID);
    }
    else
    {
      CurrentNode->SecondaryBranch = new TreeNodes(NewNodeID);
    }
    return true;
  }
  else
  {
    //try branch if it exists
    if (CurrentNode->PrimaryBranch != NULL)
    {
      if (SearchSecondaryNode(CurrentNode->PrimaryBranch,
ExistingNodeID, NewNodeID))
      {
        return true;
      }
      else
      {
        //try second branch if it exists
        if (CurrentNode->SecondaryBranch != NULL)
        {
          return(SearchSecondaryNode(CurrentNode->SecondaryBranch,
ExistingNodeID, NewNodeID));
```

```
        }
        else
        {
          return false;
        }
      }
    }
    return false;
  }
}

//Step 11
void DecisionTree::QueryTree(TreeNodes* CurrentNode)
{
  if (CurrentNode->PrimaryBranch == NULL)
  {
    //if both branches are null, tree is at a decision outcome state
    if (CurrentNode->SecondaryBranch == NULL)
    {
      //output decision 'question'
      ///////////////////////////////////////////////////////////////
///////////////////////
    }
    else
    {
      cout << "Missing Branch 1";
    }
    return;
  }
  if (CurrentNode->SecondaryBranch == NULL)
  {
    cout << "Missing Branch 2";
    return;
  }

  //otherwise test decisions at current node
  MakeDecision(CurrentNode);
}

//Step 10
void DecisionTree::Query()
{
```

```
  QueryTree(m_RootNode);
}

////////////////////////////////////////////////////////////
//debate decisions   create new function for decision logic

// cout << node->stringforquestion;

//Step 12
void DecisionTree::MakeDecision(TreeNodes *node)
{
  //should I declare variables here or inside of decisions.h
  int PHealth;
  int MHealth;
  int PStrength;
  int MStrength;
  int DistanceFBase;
  int DistanceFMonster;

  ////sets random!
  srand(time(NULL));

  //randomly create the numbers for health, strength and distance for
each variable
  PHealth = random(60);
  MHealth = random(60);
  PStrength = random(50);
  MStrength = random(50);
  DistanceFBase = random(75);
  DistanceFMonster = random(75);

  //the decision to be made string example: Player health: Monster
Health:  player health is lower/higher
  cout << "Player Health: " << PHealth << endl;
  cout << "Monster Health: " << MHealth << endl;
  cout << "Player Strength: " << PStrength << endl;
  cout << "Monster Strength: " << MStrength << endl;
  cout << "Distance Player is From Base: " << DistanceFBase << endl;
  cout << "Distance Player is From Monster: " << DistanceFMonster <<
endl;


  if (PHealth > MHealth)
  {
```

```
      std::cout << "Player health is greater than monster health";
      //Do some logic here
    }
    else
    {
      std::cout << "Monster health is greater than player health";
      //Do some logic here
    }

    if (PStrength > MStrength)
    {
      //Do some logic here
    }
    else
    {
    }

    //recursive question for next branch. Player distance from base/
  monster.
    if (DistanceFBase > DistanceFMonster)
    {
    }
    else
    {
    }

  }


  void DecisionTree::Output()
  {
    //take respective node
    DisplayTree(m_RootNode);
  }

  //Step 9
  void DecisionTree::DisplayTree(TreeNodes* CurrentNode)
  {
    //if it doesn't exist, don't display of course
    if (CurrentNode == NULL)
    {
```

```
    return;
  }

  ////////////////////////////////////////////////////////////////////
//////////////////////////////
  //need to make a string to display for each branch
  cout << "Node ID " << CurrentNode->m_NodeID << "Decision Display: "
<< endl;

  //go down branch 1
  DisplayTree(CurrentNode->PrimaryBranch);

  //go down branch 2
  DisplayTree(CurrentNode->SecondaryBranch);
}


void DecisionTree::RemoveNode(TreeNodes *node)
{

  if (node != NULL)
  {
    if (node->PrimaryBranch != NULL)
    {
      RemoveNode(node->PrimaryBranch);
    }

    if (node->SecondaryBranch != NULL)
    {
      RemoveNode(node->SecondaryBranch);
    }

    cout << "Deleting Node" << node->m_NodeID << endl;

    //delete node from memory
    delete node;
    //reset node
    node = NULL;
  }
}


int main()
{
```

```
    //create the new decision tree object
    DecisionTree* NewTree = new DecisionTree();

    //add root node   the very first 'Question' or decision to be made
    //is monster health greater than player health?
    NewTree->CreateRootNode(1);

    //add nodes depending on decisions
    //2nd decision to be made
    //is monster strength greater than player strength?
    NewTree->PrimaryNode(1, 2);

    //3rd decision
    //is the monster closer than home base?
    NewTree->SecondaryNode(1, 3);

    //depending on the weights of all three decisions, will return
  certain node result
    //results!
    //Run, Attack,
    NewTree->PrimaryNode(2, 4);
    NewTree->SecondaryNode(2, 5);
    NewTree->PrimaryNode(3, 6);
    NewTree->SecondaryNode(3, 7);


    NewTree->Output();

    //ask/answer question decision making process
    NewTree->Query();

    cout << "Decision Made. Press Any Key To Quit." << endl;

    int a;
    cin >> a;

    //release memory!
    delete NewTree;

    //return random value
    //return 1;


  }
```

How it works...

As the name suggests, a decision tree is a subset of the tree data structure. Therefore, there is a root node and two child nodes. The root node denotes a condition and the child nodes will have the probable solutions. On the next level, those solution nodes will become part of the condition, which will lead to two more solution nodes. Hence, as the preceding example shows, the entire structure is modeled on the basis of a tree structure. We have a root node and then primary and secondary nodes. We need to traverse the tree to continuously find the answers to a situation based on the root nodes and the child nodes.

We have also written a `Query` function that will query the tree structure to find out what the most probable scenario is for the situation. That in turn will get the help of a decision function, which will add its own level of heuristics, combined with the result of the query, and generate the output for the solution.

Decision trees are extremely fast, because for every scenario we are checking only half the tree. So in effect we have reduced the search space by half. The tree structure also makes it robust, so that we can add and remove nodes on the fly as well. This gives us a lot of flexibility and the overall architecture of the game is improved.

Adding behavioral movements

When we talk about AI in games, after pathfinding the next most important thing to consider is movement. When does an AI decide that it has to walk, run, jump, or slide? The ability to make these decisions quickly and correctly will make the AI really competitive in games and extremely difficult to beat. We can do all this with the help of behavioral movements.

Getting ready

For this recipe, you will need a Windows machine and Visual Studio. No other prerequisites are required.

How to do it...

In this example, you will find out how easy it is to create a decision tree. Add a source file called `Source.cpp` and add the following code to it:

```
/* Adding Behavorial Movements*/


#include <iostream>
using namespace std;
class Machine
{
```

```
    class State *current;
  public:
    Machine();
    void setCurrent(State *s)
    {
      current = s;
    }
    void Run();
    void Walk();
  };

  class State
  {
  public:
    virtual void Run(Machine *m)
    {
      cout << "   Already Running\n";
    }
    virtual void Walk(Machine *m)
    {
      cout << "   Already Walking\n";
    }
  };

  void Machine::Run()
  {
    current->Run(this);
  }

  void Machine::Walk()
  {
    current->Walk(this);
  }

  class RUN : public State
  {
  public:
    RUN()
    {
      cout << "   RUN-ctor ";
    };
    ~RUN()
    {
      cout << "   dtor-RUN\n";
```

```
    };
    void Walk(Machine *m);
  };

  class WALK : public State
  {
  public:
    WALK()
    {
      cout << "   WALK-ctor ";
    };
    ~WALK()
    {
      cout << "   dtor-WALK\n";
    };
    void Run(Machine *m)
    {
      cout << " Changing behaviour from WALK to RUN";
      m->setCurrent(new RUN());
      delete this;
    }
  };

  void RUN::Walk(Machine *m)
  {
    cout << "   Changing behaviour RUN to WALK";
    m->setCurrent(new WALK());
    delete this;
  }

  Machine::Machine()
  {
    current = new WALK();
    cout << '\n';
  }

  int main()
  {
    Machine m;
    m.Run();
    m.Walk();
    m.Walk();

    int a;
```

```
    cin >> a;

    return 0;
}
```

How it works...

In this example, we have implemented a simple state machine. The state machine is created with the **state-machine** design pattern in mind. So the states in this case are walk and run. The objective is that if the AI is walking and then needs to switch to running, it can do so at runtime. Similarly, if it is running, it can switch to walking at runtime. However, if it is already walking, and a request comes to walk, it should notify itself that there is no need to change the state.

All these change of states are handled by a class called machine, hence the name state-machine pattern. The reason why this structure is preferred by many over the traditional state machine design is that all the states need not be defined in one class and then a switch case statement can be used to change states. Although this method is correct, every additional step that is added to the game would require changing and adding to the same class structure. This is a recipe for bugs and possible disasters in the future. Instead, we are going for a more object-oriented approach where every state is a class in itself.

The `machine` class holds a pointer to the `StateTo` class and then pushes the request to the appropriate child class of the state. If we need to add the jump state, we do not need to change much in the code. We need to write a new `jump` class and add the corresponding functionalities. Because the machine has a pointer to the base class (state), it will correspondingly push the request for jump to the correct derived class.

Using neural network

Artificial neural networks (**ANNs**) are an advanced form of AI used in some games. They may not be directly used in-game; however, they may be used during the production phase to train the AI agents. Neural nets are mostly used as predictive algorithms. Based on certain parameters, and historical data, they calculate the most likely decision or attribute that the AI agent will distribute. ANNs are not restricted to games; they are used across multiple diverse domains to predict possible outcomes.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio.

How to do it...

Take a look at the following code snippet:

```
class neuralNetworkTrainer
{

private:

  //network to be trained
  neuralNetwork* NN;

  //learning parameters
  double learningRate;          // adjusts the step size of the weight
update
  double momentum;              // improves performance of stochastic
learning (don't use for batch)

  //epoch counter
  long epoch;
  long maxEpochs;

  //accuracy/MSE required
  double desiredAccuracy;

  //change to weights
  double** deltaInputHidden;
  double** deltaHiddenOutput;

  //error gradients
  double* hiddenErrorGradients;
  double* outputErrorGradients;

  //accuracy stats per epoch
  double trainingSetAccuracy;
  double validationSetAccuracy;
  double generalizationSetAccuracy;
  double trainingSetMSE;
  double validationSetMSE;
  double generalizationSetMSE;

  //batch learning flag
```

```
    bool useBatch;

    //log file handle
    bool loggingEnabled;
    std::fstream logFile;
    int logResolution;
    int lastEpochLogged;

  public:

    neuralNetworkTrainer( neuralNetwork* untrainedNetwork );
    void setTrainingParameters( double lR, double m, bool batch );
    void setStoppingConditions( int mEpochs, double dAccuracy);
    void useBatchLearning( bool flag ){ useBatch = flag; }
    void enableLogging( const char* filename, int resolution );

    void trainNetwork( trainingDataSet* tSet );


  private:
    inline double getOutputErrorGradient( double desiredValue, double
  outputValue );
    double getHiddenErrorGradient( int j );
    void runTrainingEpoch( std::vector<dataEntry*> trainingSet );
    void backpropagate(double* desiredOutputs);
    void updateWeights();
  };

  class neuralNetwork
  {

  private:

    //number of neurons
    int nInput, nHidden, nOutput;

    //neurons
    double* inputNeurons;
    double* hiddenNeurons;
    double* outputNeurons;

    //weights
    double** wInputHidden;
```

```
    double** wHiddenOutput;
    friend neuralNetworkTrainer;

  public:

    //constructor & destructor
    neuralNetwork(int numInput, int numHidden, int numOutput);
    ~neuralNetwork();

    //weight operations
    bool loadWeights(char* inputFilename);
    bool saveWeights(char* outputFilename);
    int* feedForwardPattern( double* pattern );
    double getSetAccuracy( std::vector<dataEntry*>& set );
    double getSetMSE( std::vector<dataEntry*>& set );



  private:

    void initializeWeights();
    inline double activationFunction( double x );
    inline int clampOutput( double x );
    void feedForward( double* pattern );

  };
```

How it works...

In this example snippet, we have created the backbone to write a neural network that can predict a letter which is drawn on the screen. Many devices and touch screen tablets have this ability to detect a letter that you draw on screen. Let us take this and think in terms of game design. If we want to create a game in which we draw shapes, and the corresponding weapon will be given to us, which we can then use in battle, we can use this as a template to train the agents to identify a shape before the game is released onto the market. Generally, games like these only detect basic shapes. These can be easily detected and do not require neural nets to train agents.

In games, ANNs will mostly be used to create good AI behavior. However, it is not wise to use ANNs while the game is being played, as they are expensive and take a long time to train agents. Let us look at the following example:

Class type	**Speed**	**HP**
Melee	Speed (4)	25 (HP)
Archer	Speed (7)	22 (HP)
Magic	Speed (6.4)	20 (HP)
?	Speed (6.6)	21 (HP)

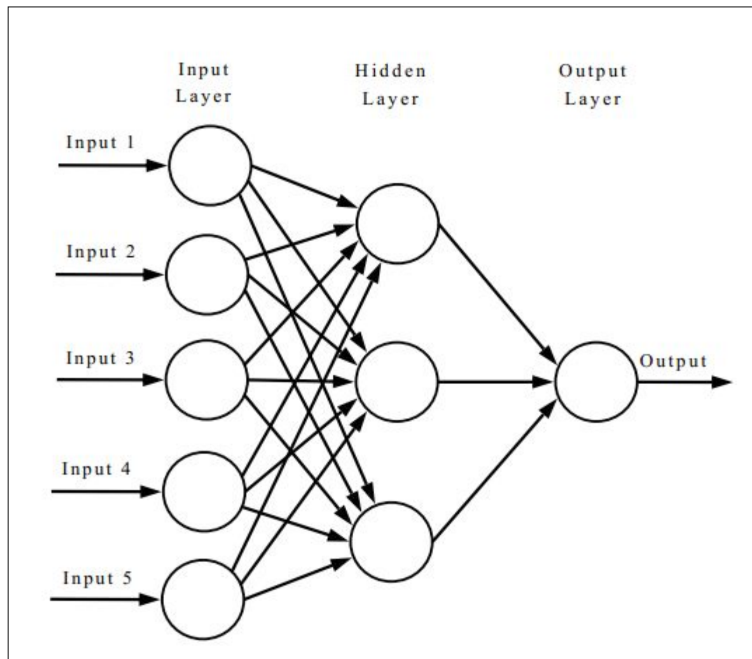
Given the data, what is the most likely class of the unknown? The number of parameters (**Class type**, **Speed**, and **HP**) is only three, but in reality it will be over 10. It will be difficult to predict the class by just looking at those numbers. That's where an ANN comes in. It can predict any of the missing column data based on other columns' data and previous historical data. This becomes a very handy tool for the designer to use to balance the game.

A few concepts of the ANN which we have implemented is necessary to understand.

An ANN is typically defined by three types of parameters:

- The interconnection pattern between the different layers of neurons.
- The learning process for updating the weights of the interconnections.
- The activation function that converts a neuron's weighted input to its output activation.

Let's take a look at the following diagram explaining the layers:



Input Layer is the layer in which we supply all the column data that is known, both historical and new. The process first involves supplying data whose output we already know. This phase is known as the learning phase. There are two types of learning algorithms, supervised and non-supervised. The explanation for these is out of the scope of this book. After that, there is a training algorithm that is applied to minimize the errors in the desired output. Back-propagation is one such technique, in which the weights that calculate the neural network function are adjusted till we get close to the desired result. After the network is set and is giving correct results for already known outputs, we can then supply new data and find the results for the unknown column data.

Using genetic algorithms

A **genetic algorithm** (**GA**) is a method of **evolutionary algorithm** (**EA**). They are particularly useful when we want to write predictive algorithms in which only the strongest is selected and the rest are rejected. This is how it gets its name. So at every iteration it mutates, does a cross-over, and only the best is selected for the next iteration of population. The idea behind genetic algorithms is that after multiple iterations only the best possible candidates are left.

Getting ready

To work through this recipe, you will need a machine running Windows with an installed version of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to write a genetic algorithm:

```
void crossover(int &seed);
void elitist();
void evaluate();
int i4_uniform_ab(int a, int b, int &seed);
void initialize(string filename, int &seed);
void keep_the_best();
void mutate(int &seed);
double r8_uniform_ab(double a, double b, int &seed);
void report(int generation);
void selector(int &seed);
void timestamp();
void Xover(int one, int two, int &seed);
```

How it works...

GA may seem extremely difficult to understand or make sense of at first. However, GAs are extremely simple. Let us think of a situation in which we have a land that is filled with dragons with different attributes. The objective or goal of the dragon is to defeat a human player who has some attributes.

Dragon(AI)

	Attribute 1	**Attribute 2**	**Attribute 3**
Dragon 1	Run	~~Defend~~	~~Attack~~
Dragon 2	~~Run~~	~~Defend~~	Attack
Dragon 3	~~Run~~	Defend	~~Attack~~

Human(Player)

	Attribute 1	**Attribute 2**	**Attribute 3**
Player	Run	Defend	Attack

So for the Dragon to be competitive against the Human, it must learn how to run, defend, and attack. Let us see how GA helps us to do this:

Step 1 (Initial Population)

Dragon(AI):

This is our initial population. Each has its own set of properties. We are just considering three dragons. In practice, there will be more than that.

	Attribute 1	Attribute 2	Attribute 3
Dragon 1	Run	~~Defend~~	~~Attack~~
Dragon 2	~~Run~~	~~Defend~~	Attack
Dragon 3	~~Run~~	Defend	~~Attack~~

Step 2 (Fitness function)

The fitness function (%) determines how fit a particular dragon is from the population. 100% is perfect fitness.

Fitness	Dragon	Attribute 1	Attribute 2	Attribute 3
60%	Dragon 1	Run	~~Defend~~	~~Attack~~
75%	Dragon 2	~~Run~~	~~Defend~~	Attack
20%	Dragon 3	~~Run~~	Defend	~~Attack~~

Step 3 Cross-over

Based on the fitness function and the attributes that are missing, there will be a cross-over or reproduction phase to create a new dragon with both properties:

Table 1

Fitness	Dragon	Attribute 1	Attribute 2	Attribute 3
60%	Dragon 1	Run	Defend	Attack
75%	Dragon 2	Run	Defend	Attack
20%	Dragon 3	Run	Defend	Attack

Table 2

Fitness	Dragon	Attribute 1	Attribute 2	Attribute 3
60%	Dragon 1	Run	~~Defend~~	~~Attack~~
75%	Dragon 2	~~Run~~	~~Defend~~	Attack
20%	Dragon 3	~~Run~~	Defend	~~Attack~~

The dragon with the least fitness function will be removed from the population. (Survival of the fittest).

Fitness	Dragon	Attribute 1	Attribute 2	Attribute 3
60%	Dragon 1	Run	~~Defend~~	~~Attack~~
75%	Dragon 2	~~Run~~	~~Defend~~	Attack
~~*20%*~~	~~*Dragon 3*~~	~~*Run*~~	~~*Defend*~~	~~*Attack*~~

Step 4 Mutate

So we have now got a new dragon that can run as well as attack and has a fitness function of *67%*:

Fitness	Dragon	Attribute 1	Attribute 2	Attribute 3
67%	Dragon 4	Run	~~Defend~~	Attack

We must now repeat the process (new generation) with other dragons in the population until we are satisfied with the result. The ideal population will be when all dragons have the following capabilities:

Fitness	Dragon	Attribute 1	Attribute 2	Attribute 3
100%	Dragon 4	Run	Defend	Attack

However, this may not always be possible. We need to be satisfied it is closer to the goal. All the stages described here are implemented as functions, and could be expanded upon based on the requirements of the AI agent.

Now you could ask, why don't we create dragons with all the properties in the first place? That's where adaptive AI comes into play. If we define all the properties in the dragons before the user plays the game, it may be very easy to defeat the dragons as the game progresses. However, if the AI dragons can adapt based on how the player defeats them, it may get progressively more difficult to beat the AI. As the player defeats the AI, we need to record the parameters and add that parameter as a goal attribute for the dragon, which it can achieve after a few cross-overs and mutations.

Using other waypoint systems

Waypoints are a way of writing pathfinding algorithms. They are extremely easy to write. However, if not thought out properly, they can be extremely buggy and the AI can look extremely stupid. Many older games often had this sort of bug, which resulted in a revolution in the implementation of waypoint systems.

Getting ready

To work through this recipe, you will need a machine running Windows with an installed version of Visual Studio. No other prerequisites are required.

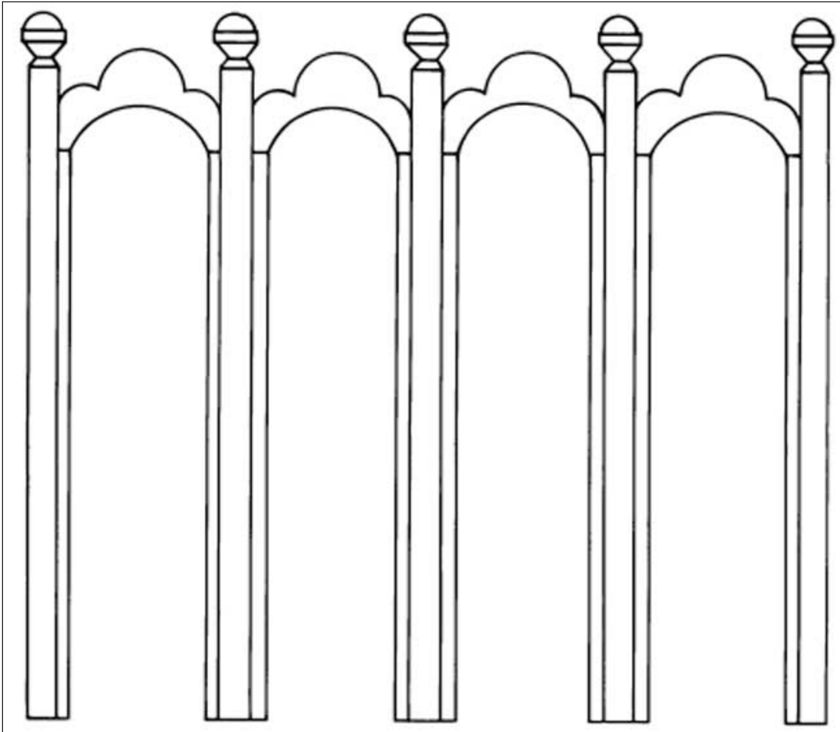
How to do it...

In this recipe, we will find out how easy it is to create waypoint systems:

```
#include <iostream>

using namespace std;

int main()
{
  float positionA = 4.0f; float positionB = 2.0f; float positionC =
-1.0f; float positionD = 10.0f; float positionE = 0.0f;

  //Sort the points according to Djisktra's
  //A* can be used on top of this to minimise the points for traversal
  //Transform the  objects over these new points.
  return 0;
}
```

How it works...

In this example, we will just discuss a basic implementation of the waypoint system. As the name suggests, waypoints are just 2D/3D points in world space that we want the AI agent to follow. All the agent has to do is move from point **A** to point **B**. However, this has complications. For example, let us consider the following diagram:



To get from **A** to **B** is easy. Now, to get from **B** to **C** it has to follow a pathfinding algorithm such as A* or Djikstra's algorithm. In that case, it will avoid the obstacle in the center and move towards **C**. Now let's say it has suddenly seen the user at point **A**, part way through the journey. How should it react? If we just provide waypoints, it will look at its dictionary of points that it is allowed to move to and which is closest to it. The answer will be **A**. However, if it starts going towards **A**, it will be blocked by the wall and it may get stuck in a loop, hitting the wall continuously. You may have seen this behavior a lot in older games. In this case, the AI must make a decision to go back to **B** and then to **A**. So we can't use a waypoint algorithm on its own. For better performance and efficiency, we need to write a decision-making algorithm and a pathfinding algorithm along with it. This is what is used in most modern games, along with techniques such as **NavMesh** and so on.

9
Physics in Game Development

In this chapter, the following recipes will be covered:

- Using physics rules in your game
- Making things collide
- Installing and integrating Box2D
- Making a basic 2D game
- Making a 3D game
- Creating a particle system
- Using ragdoll in your game

Introduction

In modern games, and games of the past, some type of physics has always been added to increase the sense of realism. Although most physics in games is an approximation or optimization of actual physics rules, it does a good job of achieving the desired results. Physics in games is basically a rough implementation of the Newtonian laws of motion, mixed with the basic fundamentals of collision detection.

The trick for a games developer is to write the code in such a way that it does not bottleneck the CPU and the game still runs at a desired framework. We will discuss some basic concepts that we require to introduce physics into our game. For the sake of simplicity, we have integrated **Box2D** into our engine and, along with a renderer (**OpenGL**), we will output some physics interaction between objects. For 3D physics, we will get help from the **Bullet Physics** SDK and display the desired result.

Using physics rules in your game

The first step to have physics in the game is to have the environment ready so that proper calculations can be applied to the bodies, and the physics simulation can work on them.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to add physics rules to our game:

1. First, set up all the objects in the game scene.
2. Give them properties so that they have vector points and velocities.
3. Assign bounding boxes or bounding circles, depending on the shape of the object.
4. Apply forces on each of the bodies.
5. Detect collisions between them based on the shape.
6. Solve for the constraints.
7. Output the result.

Take a look at the following code snippet:

```
#include <Box2D/Collision/b2Collision.h>
#include <Box2D/Collision/Shapes/b2CircleShape.h>
#include <Box2D/Collision/Shapes/b2PolygonShape.h>

void b2CollideCircles(
  b2Manifold* manifold,
  const b2CircleShape* circleA, const b2Transform& xfA,
  const b2CircleShape* circleB, const b2Transform& xfB)
{
  manifold->pointCount = 0;

  b2Vec2 pA = b2Mul(xfA, circleA->m_p);
  b2Vec2 pB = b2Mul(xfB, circleB->m_p);

  b2Vec2 d = pB - pA;
  float32 distSqr = b2Dot(d, d);
  float32 rA = circleA->m_radius, rB = circleB->m_radius;
  float32 radius = rA + rB;
```

```
  if (distSqr > radius * radius)
  {
    return;
  }

  manifold->type = b2Manifold::e_circles;
  manifold->localPoint = circleA->m_p;
  manifold->localNormal.SetZero();
  manifold->pointCount = 1;

  manifold->points[0].localPoint = circleB->m_p;
  manifold->points[0].id.key = 0;
}

void b2CollidePolygonAndCircle(
  b2Manifold* manifold,
  const b2PolygonShape* polygonA, const b2Transform& xfA,
  const b2CircleShape* circleB, const b2Transform& xfB)
{
  manifold->pointCount = 0;

  // Compute circle position in the frame of the polygon.
  b2Vec2 c = b2Mul(xfB, circleB->m_p);
  b2Vec2 cLocal = b2MulT(xfA, c);

  // Find the min separating edge.
  int32 normalIndex = 0;
  float32 separation = -b2_maxFloat;
  float32 radius = polygonA->m_radius + circleB->m_radius;
  int32 vertexCount = polygonA->m_count;
  const b2Vec2* vertices = polygonA->m_vertices;
  const b2Vec2* normals = polygonA->m_normals;

  for (int32 i = 0; i < vertexCount; ++i)
  {
    float32 s = b2Dot(normals[i], cLocal - vertices[i]);

    if (s > radius)
    {
      // Early out.
      return;
    }

    if (s > separation)
```

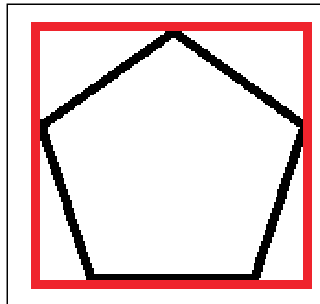
```
        {
          separation = s;
          normalIndex = i;
        }
      }

      // Vertices that subtend the incident face.
      int32 vertIndex1 = normalIndex;
      int32 vertIndex2 = vertIndex1 + 1 < vertexCount ? vertIndex1 + 1 :
    0;
      b2Vec2 v1 = vertices[vertIndex1];
      b2Vec2 v2 = vertices[vertIndex2];

      // If the center is inside the polygon ...
      if (separation < b2_epsilon)
      {
        manifold->pointCount = 1;
        manifold->type = b2Manifold::e_faceA;
        manifold->localNormal = normals[normalIndex];
        manifold->localPoint = 0.5f * (v1 + v2);
        manifold->points[0].localPoint = circleB->m_p;
        manifold->points[0].id.key = 0;
        return;
      }

      // Compute barycentric coordinates
      float32 u1 = b2Dot(cLocal - v1, v2 - v1);
      float32 u2 = b2Dot(cLocal - v2, v1 - v2);
      if (u1 <= 0.0f)
      {
        if (b2DistanceSquared(cLocal, v1) > radius * radius)
        {
          return;
        }

        manifold->pointCount = 1;
        manifold->type = b2Manifold::e_faceA;
        manifold->localNormal = cLocal - v1;
        manifold->localNormal.Normalize();
        manifold->localPoint = v1;
        manifold->points[0].localPoint = circleB->m_p;
        manifold->points[0].id.key = 0;
      }
```

```
  else if (u2 <= 0.0f)
  {
    if (b2DistanceSquared(cLocal, v2) > radius * radius)
    {
      return;
    }

    manifold->pointCount = 1;
    manifold->type = b2Manifold::e_faceA;
    manifold->localNormal = cLocal - v2;
    manifold->localNormal.Normalize();
    manifold->localPoint = v2;
    manifold->points[0].localPoint = circleB->m_p;
    manifold->points[0].id.key = 0;
  }
  else
  {
    b2Vec2 faceCenter = 0.5f * (v1 + v2);
    float32 separation = b2Dot(cLocal - faceCenter,
normals[vertIndex1]);
    if (separation > radius)
    {
      return;
    }

    manifold->pointCount = 1;
    manifold->type = b2Manifold::e_faceA;
    manifold->localNormal = normals[vertIndex1];
    manifold->localPoint = faceCenter;
    manifold->points[0].localPoint = circleB->m_p;
    manifold->points[0].id.key = 0;
  }
}
```

How it works...

The first step for a body to exhibit physics properties is for it to be a rigid body. This is however not true if your body is supposed to have some kind of fluid physics, as is the case for a plastic or any other soft body. In that case, we will have to set up the world differently, as it is a far more complex problem. A rigid body, in short, is any object in world space that will not deform, even if external sources are applied to it. Even in game engines such as Unity or UE4, if you assign a body as a rigid body, it will automatically react, based on the physics simulation property of the engine. After the rigid body is set up, we need to determine if the body is static or dynamic. This step is important as we can greatly reduce the number of calculations if we know that the body is static. A dynamic body must be assigned velocities as well as vector positions.

After the previous step is complete, the next step is to add colliders or bounding objects. These will actually be used for the calculation of collision points. For example, if we have a 3D model of a human, it is sometimes not very wise to use the exact body mesh for collisions. Instead, we could use a capsule, which is a cylinder with two half spheres on either end for the body and a similar structure for the hands. In the case of a 2D object, we make a choice between a circular bounding object or a box bounding object. The following diagram shows the object in black and the bounding box in red. We can now apply force or impulse to the objects:



The next step in the pipeline is to actually detect when two objects have collided. We will discuss this further in the next recipe. But let's say we have to detect whether `circle A` has collided with `circle B`; in most cases we just need information on whether they have collided, rather than the exact point of contact. In this case, we need to write some mathematical functions to detect that. We then return the output and, based on that, we write our logic for collision and finally display the result.

In the preceding example, there is a function called `b2CollidePolygonAndCircle` which is used to calculate the collision between a polygon and a circle. We define the two shapes and then try to calculate various details that determine whether the points of the polygon and the circle intersect. We need to find the edge list point and then calculate whether the points lie inside the other shape, and so on.

Making things collide

A huge part of the physics system is making things collide. We need to figure out whether the objects have collided and pass on the relevant information. In this recipe, we will look at different techniques to do this.

Getting ready

You need a Windows machine and a working copy of Visual Studio. No other prerequisites are required.

How to do it…

In this recipe, we will find out how easy it is to detect collisions:

```
#include <Box2D/Collision/b2Collision.h>
#include <Box2D/Collision/Shapes/b2PolygonShape.h>

// Find the max separation between poly1 and poly2 using edge normals
from poly1.
static float32 b2FindMaxSeparation(int32* edgeIndex,
             const b2PolygonShape* poly1, const b2Transform& xf1,
             const b2PolygonShape* poly2, const b2Transform& xf2)
{
  int32 count1 = poly1->m_count;
  int32 count2 = poly2->m_count;
  const b2Vec2* n1s = poly1->m_normals;
  const b2Vec2* v1s = poly1->m_vertices;
  const b2Vec2* v2s = poly2->m_vertices;
  b2Transform xf = b2MulT(xf2, xf1);

  int32 bestIndex = 0;
  float32 maxSeparation = -b2_maxFloat;
  for (int32 i = 0; i < count1; ++i)
  {
    // Get poly1 normal in frame2.
    b2Vec2 n = b2Mul(xf.q, n1s[i]);
    b2Vec2 v1 = b2Mul(xf, v1s[i]);

    // Find deepest point for normal i.
    float32 si = b2_maxFloat;
    for (int32 j = 0; j < count2; ++j)
    {
```

```
        float32 sij = b2Dot(n, v2s[j] - v1);
        if (sij < si)
        {
          si = sij;
        }
      }

      if (si > maxSeparation)
      {
        maxSeparation = si;
        bestIndex = i;
      }
    }

    *edgeIndex = bestIndex;
    return maxSeparation;
  }

  static void b2FindIncidentEdge(b2ClipVertex c[2],
             const b2PolygonShape* poly1, const b2Transform& xf1, int32
  edge1,
             const b2PolygonShape* poly2, const b2Transform& xf2)
  {
    const b2Vec2* normals1 = poly1->m_normals;

    int32 count2 = poly2->m_count;
    const b2Vec2* vertices2 = poly2->m_vertices;
    const b2Vec2* normals2 = poly2->m_normals;

    b2Assert(0 <= edge1 && edge1 < poly1->m_count);

    // Get the normal of the reference edge in poly2's frame.
    b2Vec2 normal1 = b2MulT(xf2.q, b2Mul(xf1.q, normals1[edge1]));

    // Find the incident edge on poly2.
    int32 index = 0;
    float32 minDot = b2_maxFloat;
    for (int32 i = 0; i < count2; ++i)
    {
      float32 dot = b2Dot(normal1, normals2[i]);
      if (dot < minDot)
      {
        minDot = dot;
        index = i;
```

```
    }
  }

  // Build the clip vertices for the incident edge.
  int32 i1 = index;
  int32 i2 = i1 + 1 < count2 ? i1 + 1 : 0;

  c[0].v = b2Mul(xf2, vertices2[i1]);
  c[0].id.cf.indexA = (uint8)edge1;
  c[0].id.cf.indexB = (uint8)i1;
  c[0].id.cf.typeA = b2ContactFeature::e_face;
  c[0].id.cf.typeB = b2ContactFeature::e_vertex;

  c[1].v = b2Mul(xf2, vertices2[i2]);
  c[1].id.cf.indexA = (uint8)edge1;
  c[1].id.cf.indexB = (uint8)i2;
  c[1].id.cf.typeA = b2ContactFeature::e_face;
  c[1].id.cf.typeB = b2ContactFeature::e_vertex;
}

// Find edge normal of max separation on A - return if separating axis
is found
// Find edge normal of max separation on B - return if separation axis
is found
// Choose reference edge as min(minA, minB)
// Find incident edge
// Clip

// The normal points from 1 to 2
void b2CollidePolygons(b2Manifold* manifold,
            const b2PolygonShape* polyA, const b2Transform& xfA,
            const b2PolygonShape* polyB, const b2Transform& xfB)
{
  manifold->pointCount = 0;
  float32 totalRadius = polyA->m_radius + polyB->m_radius;

  int32 edgeA = 0;
  float32 separationA = b2FindMaxSeparation(&edgeA, polyA, xfA, polyB,
xfB);
  if (separationA > totalRadius)
    return;

  int32 edgeB = 0;
```

```
  float32 separationB = b2FindMaxSeparation(&edgeB, polyB, xfB, polyA,
xfA);
  if (separationB > totalRadius)
    return;

  const b2PolygonShape* poly1;  // reference polygon
  const b2PolygonShape* poly2;  // incident polygon
  b2Transform xf1, xf2;
  int32 edge1;          // reference edge
  uint8 flip;
  const float32 k_tol = 0.1f * b2_linearSlop;

  if (separationB > separationA + k_tol)
  {
    poly1 = polyB;
    poly2 = polyA;
    xf1 = xfB;
    xf2 = xfA;
    edge1 = edgeB;
    manifold->type = b2Manifold::e_faceB;
    flip = 1;
  }
  else
  {
    poly1 = polyA;
    poly2 = polyB;
    xf1 = xfA;
    xf2 = xfB;
    edge1 = edgeA;
    manifold->type = b2Manifold::e_faceA;
    flip = 0;
  }

  b2ClipVertex incidentEdge[2];
  b2FindIncidentEdge(incidentEdge, poly1, xf1, edge1, poly2, xf2);

  int32 count1 = poly1->m_count;
  const b2Vec2* vertices1 = poly1->m_vertices;

  int32 iv1 = edge1;
  int32 iv2 = edge1 + 1 < count1 ? edge1 + 1 : 0;

  b2Vec2 v11 = vertices1[iv1];
```

```
  b2Vec2 v12 = vertices1[iv2];

  b2Vec2 localTangent = v12 - v11;
  localTangent.Normalize();

  b2Vec2 localNormal = b2Cross(localTangent, 1.0f);
  b2Vec2 planePoint = 0.5f * (v11 + v12);

  b2Vec2 tangent = b2Mul(xf1.q, localTangent);
  b2Vec2 normal = b2Cross(tangent, 1.0f);

  v11 = b2Mul(xf1, v11);
  v12 = b2Mul(xf1, v12);

  // Face offset.
  float32 frontOffset = b2Dot(normal, v11);

  // Side offsets, extended by polytope skin thickness.
  float32 sideOffset1 = -b2Dot(tangent, v11) + totalRadius;
  float32 sideOffset2 = b2Dot(tangent, v12) + totalRadius;

  // Clip incident edge against extruded edge1 side edges.
  b2ClipVertex clipPoints1[2];
  b2ClipVertex clipPoints2[2];
  int np;

  // Clip to box side 1
  np = b2ClipSegmentToLine(clipPoints1, incidentEdge, -tangent,
sideOffset1, iv1);

  if (np < 2)
    return;

  // Clip to negative box side 1
  np = b2ClipSegmentToLine(clipPoints2, clipPoints1,  tangent,
sideOffset2, iv2);

  if (np < 2)
  {
    return;
  }

  // Now clipPoints2 contains the clipped points.
```

```
    manifold->localNormal = localNormal;
    manifold->localPoint = planePoint;

    int32 pointCount = 0;
    for (int32 i = 0; i < b2_maxManifoldPoints; ++i)
    {
      float32 separation = b2Dot(normal, clipPoints2[i].v) -
  frontOffset;

      if (separation <= totalRadius)
      {
        b2ManifoldPoint* cp = manifold->points + pointCount;
        cp->localPoint = b2MulT(xf2, clipPoints2[i].v);
        cp->id = clipPoints2[i].id;
        if (flip)
        {
          // Swap features
          b2ContactFeature cf = cp->id.cf;
          cp->id.cf.indexA = cf.indexB;
          cp->id.cf.indexB = cf.indexA;
          cp->id.cf.typeA = cf.typeB;
          cp->id.cf.typeB = cf.typeA;
        }
        ++pointCount;
      }
    }

    manifold->pointCount = pointCount;
  }
```

How it works...

Assuming the objects in the scene are already set up as rigid body, and the proper impulses are added to each, the next step is to detect collisions. An impulse is a force that acts on a body. The force acts briefly on the body and results in some change of momentum.

In games, collision detection usually happens in two phases. The first phase is called the **broad-phase collision** and the next phase is called the **narrow-phase collision**. The broad phase is less expensive, as it deals with the concept of which bodies are most likely to collide. The narrow phase is more expensive because it actually compares each body for collisions. In a games environment, it is not feasible to have everything in the narrow phase. Hence, most of the work is done in the broad phase. Broad phase algorithms work with sweep and prune (sort and prune) or Space partition trees. In the sweep and prune technique, all the lower ends and upper ends of the bounding boxes of a solid are sorted and checked for intersections. After that, it is sent to a more detailed check in the narrow phase. So in this method, we need to update the bounding box of the solid every time it changes its orientation. The other technique used is **BSP**. We have already discussed BSP in previous chapters. We need to partition the scene in such a way that in each subdivision, only a certain number of objects can collide. In the narrow phase collision, a more pixel-perfect collision detection algorithm is applied.

There are various ways to check for collisions. It entirely depends on the shape that is acting as the bounding box. Also, it is important to understand how the bounding box is aligned. In a normal scenario, a bounding box would be axes-aligned and would be referred to as `AABB`. To detect whether two Box2D bounding boxes have collided, we would have to do the following:

```
bool BoxesIntersect(const Box2D &a, const Box2D &b)
{
    if (a.max.x < b.min.x) return false; // a is left of b
    if (a.min.x > b.max.x) return false; // a is right of b
    if (a.max.y < b.min.y) return false; // a is above b
    if (a.min.y > b.max.y) return false; // a is below b
    return true; // boxes overlap
}
```

We can then extend this to detect more complex shapes for rectangles, circles, lines, and other polygons. If we are writing our own 2D physics engine, then we would have to write a function for each combination of shapes intersecting with one another. If we use a physics engine such as Box2D or PhysX, these functions would already be written for us and we would have to just use them properly and consistently.

Installing and integrating Box2D

To be able to work with 2D physics, one great open source physics engine is Box2D. This comes with lots of functions that are common for any 2D game built in, so we do not have to reinvent the wheel and write them again.

Getting ready

You need to have a working Windows machine.

How to do it...

Go through the following steps:

1. Go to `http://box2d.org/`.
2. Browse to `http://box2d.org/downloads/`.
3. Download or clone the latest copy from GitHub.
4. Build the solution in your version of Visual Studio. Some projects may not work, as they were built in different versions of Visual Studio.
5. If this throws an error, clean the solution, delete the `bin` folder, and rebuild it.
6. After the solution rebuilds successfully, run the `TestBed` projects.
7. If you can run the application successfully, Box2D has been integrated.

How it works...

Box2D is a physics engine built entirely in C++. As it has given us access to the source code, it means we can build it from scratch as well, and check for ourselves how each function is written. As the project is hosted on GitHub, every time a new development is made, we can clone it and be updated with all the latest code.

In the solution, Box2D already has a project called `TestBed`, which has loads of sample applications that can be run. It is actually a collection of loads of different types of application. `Test Entries` is the entry point of all the applications. It is a long array of the different applications that we want rendered in the `TestBed` project. The array contains the name of the application and the static function to initialize the world.

Finally, the output of the physics simulation is fed to the renderer, which in this case is OpenGL, and it draws the scene for us.

Making a basic 2D game

Every 2D game is different. However, we can generalize the physics functions that are going to be used in most 2D games. In this recipe, we will create a basic scene using Box2D's built-in functions and the `TestBed` project. The scene will mimic one of the most popular 2D games of our times, *Angry Birds*TM.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to add a barebones architecture for a 2D game using Box2D:

```
class Tiles : public Test
{
public:
  enum
  {
    e_count = 10
  };

  Tiles()
  {
    m_fixtureCount = 0;
    b2Timer timer;

    {
      float32 a = 1.0f;
      b2BodyDef bd;
      bd.position.y = -a;
      b2Body* ground = m_world->CreateBody(&bd);

#if 1
      int32 N = 200;
      int32 M = 10;
      b2Vec2 position;
      position.y = 0.0f;
      for (int32 j = 0; j < M; ++j)
      {
        position.x = -N * a;
        for (int32 i = 0; i < N; ++i)
        {
          b2PolygonShape shape;
          shape.SetAsBox(a, a, position, 0.0f);
          ground->CreateFixture(&shape, 0.0f);
          ++m_fixtureCount;
```

```
          position.x += 2.0f * a;
        }
        position.y -= 2.0f * a;
      }
#else
      int32 N = 200;
      int32 M = 10;
      b2Vec2 position;
      position.x = -N * a;
      for (int32 i = 0; i < N; ++i)
      {
        position.y = 0.0f;
        for (int32 j = 0; j < M; ++j)
        {
          b2PolygonShape shape;
          shape.SetAsBox(a, a, position, 0.0f);
          ground->CreateFixture(&shape, 0.0f);
          position.y -= 2.0f * a;
        }
        position.x += 2.0f * a;
      }
#endif
    }

    {
      float32 a = 1.0f;
      b2PolygonShape shape;
      shape.SetAsBox(a, a);


      b2Vec2 x(-7.0f, 0.75f);
      b2Vec2 y;
      b2Vec2 deltaX(1.125f, 2.5f);
      b2Vec2 deltaY(2.25f, 0.0f);

      for (int32 i = 0; i < e_count; ++i)
      {
        y = x;

        for (int32 j = i; j < e_count; ++j)
        {
          b2BodyDef bd;
          bd.type = b2_dynamicBody;
```

```
          bd.position = y;



          b2Body* body = m_world->CreateBody(&bd);
          body->CreateFixture(&shape, 5.0f);
          ++m_fixtureCount;
          y += deltaY;
        }

        x += deltaX;
      }
    }

    m_createTime = timer.GetMilliseconds();
  }

  void Step(Settings* settings)
  {
    const b2ContactManager& cm = m_world->GetContactManager();
    int32 height = cm.m_broadPhase.GetTreeHeight();
    int32 leafCount = cm.m_broadPhase.GetProxyCount();
    int32 minimumNodeCount = 2 * leafCount - 1;
    float32 minimumHeight = ceilf(logf(float32(minimumNodeCount)) /
logf(2.0f));
    g_debugDraw.DrawString(5, m_textLine, "dynamic tree height = %d,
min = %d", height, int32(minimumHeight));
    m_textLine += DRAW_STRING_NEW_LINE;

    Test::Step(settings);

    g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms,
fixture count = %d",
      m_createTime, m_fixtureCount);
    m_textLine += DRAW_STRING_NEW_LINE;



  }

  static Test* Create()
  {
```

```
        return new Tiles;
      }

      int32 m_fixtureCount;
      float32 m_createTime;
    };

    #endif
```

How it works...

In this example, we are using the Box2D engine to calculate the physics. The main class of `Test Entries`, as described previously, is used to store the name of the application and the static create method. In this case, the name of the application is `Tiles`. In the tiles application, we have created a physics world using Box2D shapes and functions. The pyramid of tiles is created with the help of boxes. These boxes are dynamic in nature, which means they will react and move based on the forces applied to them. The base or the ground is also made of tiles. However, those tiles are stationary and do not move. We assign a position and velocity for all the tiles that make up the ground and the pyramid. It is not practical to individually assign a position and velocity to each tile, so we do this with an iteration loop.

After the scene is built, we can interact with the pyramid using a mouse click. From the GUI, other properties can also be switched on or off. Pressing the Space bar also triggers a ball at a random position which will destroy the formation of the tiles, much like *Angry Birds*. We can also write logic to make all the tiles that collide with the ground disappear and add points to the score every time that happens, and then we have ourselves a small 2D *Angry Birds* clone.

Making a 3D game

Not much changes when we shift our focus from physics in 2D to physics in 3D. We now need to worry about another dimension. As mentioned in the previous recipes, we still need to maintain the environment so that it follows Newtonian rules and solves constraints. There are a lot of things that can go wrong while rotating the body in 3D space. In this recipe, we will look at a very basic implementation of 3D physics using the Bullet Engine SDK.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio.

How to do it...

In this recipe, we will see how easy it is to write a physics world in 3D.

For broad-phase collision take a look at the following snippet:

```
void  b3DynamicBvhBroadphase::getAabb(int objectId,b3Vector3& aabbMin,
b3Vector3& aabbMax ) const
{
  const b3DbvtProxy*            proxy=&m_proxies[objectId];
  aabbMin = proxy->m_aabbMin;
  aabbMax = proxy->m_aabbMax;
}
```

For narrow-phase collision, see the following code:

```
void b3CpuNarrowPhase::computeContacts(b3AlignedObjectArray<b3Int4>&
pairs, b3AlignedObjectArray<b3Aabb>& aabbsWorldSpace, b3AlignedObjectA
rray<b3RigidBodyData>& bodies)
{
  int nPairs = pairs.size();
  int numContacts = 0;
  int maxContactCapacity = m_data->m_config.m_maxContactCapacity;
  m_data->m_contacts.resize(maxContactCapacity);

  for (int i=0;i<nPairs;i++)
  {
    int bodyIndexA = pairs[i].x;
    int bodyIndexB = pairs[i].y;
    int collidableIndexA = bodies[bodyIndexA].m_collidableIdx;
    int collidableIndexB = bodies[bodyIndexB].m_collidableIdx;

    if (m_data->m_collidablesCPU[collidableIndexA].m_shapeType ==
SHAPE_SPHERE &&
      m_data->m_collidablesCPU[collidableIndexB].m_shapeType == SHAPE_
CONVEX_HULL)
    {
//      computeContactSphereConvex(i,bodyIndexA,bodyIndexB,collidableI
ndexA,collidableIndexB,&bodies[0],
//        &m_data->m_collidablesCPU[0],&hostConvexData[0],
&hostVertices[0],&hostIndices[0],&hostFaces[0],&hostContacts[0],
nContacts,maxContactCapacity);
    }

    if (m_data->m_collidablesCPU[collidableIndexA].m_shapeType ==
SHAPE_

  m_data->m_contacts.resize(numContacts);


<. . . . . . .  More code to follow . . . . . . . .>
}
```

How it works...

As we can see from the example above, even in 3D the physics collision system has to be divided into phases: the broad phase and the narrow phase. In a broad-phase collision, we now take into account Vector3, instead of just two float points, as we now have three axes (*x*, *y*, and *z*). We need to enter the object ID and then check within the bounds of the bounding boxes. Similarly, for a narrow-phase collision, our problem domain and calculations remain the same. We now change it to support 3D. The previous example shows a part of a problem that would arise if we need to find the contact points in a narrow phase collision. We create an array and, based on the collision callbacks, we save out all the points that are in contact. Later on, we can write other methods to check whether the points are overlapping or not.

Creating a particle system

Particle systems are quite important in games in order to add to the visual representation of the whole feel of the game. Particle systems are quite easy to write and are merely a collection of one or more particles. So we need to create a single particle with some properties and then let the particle system decide how many particles it wants.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio.

How to do it...

Add a source file called `Source.cpp`. Then add the following code to it:

```
class Particle

{
  Vector3 location;
  Vector3 velocity;
  Vector3 acceleration;
  float lifespan;

  Particle(Vector3 vec)
  {

    acceleration = new Vector3(.05, 0.05);
    velocity = new Vector3(random(-3, 3), random(-4, 0));
    location = vec.get();
```

```
    lifespan = 125.0;
  }


    void run()
    {
    update();
    display();
    }

  void update() {
    velocity.add(acceleration);
    location.add(velocity);
    lifespan -= 2.0;
  }

  void display()
  {
    stroke(0, lifespan);
    fill(0, lifespan);
    trapezoid(location.x, location.y, 8, 8);
  }


    boolean isDead()
    {
    if (lifespan < 0.0) {
      return true;
    }
    else {
      return false;
    }
  }
};

Particle p;

void setup()
{
  size(800, 600);
  p = new Particle(new Vector3(width / 2, 10));
}

void draw()
```

```
{
  for (int i = 0; i < particles.size(); i++) {
    Particle p = particles.get(i);
    p.run();

      if (p.isDead()) {
        particles.remove(i);
      }
  }
}
```

How it works...

As we can see in the example, our first task is to create a `particle` class. The `particle` class will have properties such as `velocity`, `acceleration`, `position`, and `lifespan`. Because we are making the particle in 3D space, we are using Vector3 to denote the particle's properties. If we were to create the particle in 2D space, we would have used Vector2 to do this. In the constructor, we assign the starting values of the attributes. We then have two main functions, `update` and `display`. The `update` function updates the `velocity` and `position` every frame, and also reduces the lifespan so that it disappears when its lifespan is over. In the `display` function, we need to specify how we want the particle to be viewed: whether it should have stroke or fill, and so on. Here we also have to specify the shape of the particle. The most common shape is a sphere or a cone. We have used a trapezoid just to denote that we can specify any shape. Finally, from the client program, we need to call this object and then access the various functions to display the particle.

However, all this will do is display just one particle on the screen. Of course, we can create an array of 100 objects and that would display 100 particles on the screen. A better approach is to create a particle system, which creates an array of particles. The number of particles that will be drawn is specified by the client program. Based on the request, the particle system draws the required number of particles. Also, there must be a function to determine which particles are to be removed from the screen. This is dependent on the lifespan of each particle.

Using ragdoll in your game

Ragdoll physics is a special kind of procedural animation that is often used as a replacement for traditional static death animations in games. The whole idea of ragdoll animation is that after death a character falls as if the bones of the body are behaving like a ragdoll. Hence the name. It has nothing to do with realism, but adds a special fun element to the game.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio. The DirectX SDK is also required; preferably use the DirectX June 2010 edition.

How to do it...

Let us take a look at the following code:

```
#include "RagDoll.h"
#include "C3DETransform.h"
#include "PhysicsFactory.h"
#include "Physics.h"
#include "DebugMemory.h"

RagDoll::RagDoll(C3DESkinnedMesh * a_skinnedMesh,
C3DESkinnedMeshContainer * a_skinnedMeshContainer, int totalParts, int
totalConstraints)
{
  m_skinnedMesh = a_skinnedMesh;
  m_skinnedMeshContainer = a_skinnedMeshContainer;
  m_totalParts = totalParts;
  m_totalConstraints = totalConstraints;

  m_ragdollBodies = (btRigidBody**)malloc(sizeof(btRigidBody) *
totalParts);
  m_ragdollShapes = (btCollisionShape**)
malloc(sizeof(btCollisionShape) * totalParts);
  m_ragdollConstraints = (btTypedConstraint**)malloc(sizeof(btTypedCon
straint) * totalConstraints);



  m_boneIndicesToFollow = (int*) malloc(sizeof(int) * m_skinnedMesh-
>GetTotalBones());

  m_totalBones = m_skinnedMesh->GetTotalBones();

  m_bonesCurrentWorldPosition = (D3DXMATRIX**)
malloc(sizeof(D3DXMATRIX) * m_totalBones);

  m_boneToPartTransforms = (D3DXMATRIX**)malloc(sizeof(D3DXMATRIX) *
m_totalBones);


  for(int i = 0; i < totalConstraints; i++)
```

```
    {
      m_ragdollConstraints[i] = NULL;
    }

    for(int i = 0; i < totalParts; i++)
    {
      m_ragdollBodies[i] = NULL;
      m_ragdollShapes[i] = NULL;

    }

    for(int i = 0; i < m_totalBones; i++)
    {
      m_boneToPartTransforms[i] = NULL;
      m_boneToPartTransforms[i] = new D3DXMATRIX();

      m_bonesCurrentWorldPosition[i] = NULL;
      m_bonesCurrentWorldPosition[i] = new D3DXMATRIX();
    }


    m_constraintCount = 0;
  }

  RagDoll::~RagDoll()
  {
    free(m_ragdollConstraints);
    free(m_ragdollBodies);
    free(m_ragdollShapes);

    for(int i = 0; i < m_totalBones; i++)
    {


      delete m_boneToPartTransforms[i];
      m_boneToPartTransforms[i] = NULL;

      delete m_bonesCurrentWorldPosition[i];
      m_bonesCurrentWorldPosition[i] = NULL;
    }

    free(m_bonesCurrentWorldPosition);
    free(m_boneToPartTransforms);
```

```
  free(m_boneIndicesToFollow);

}

int RagDoll::GetTotalParts()
{
  return m_totalParts;
}

int RagDoll::GetTotalConstraints()
{
  return m_totalConstraints;
}


C3DESkinnedMesh *RagDoll::GetSkinnedMesh()
{
  return m_skinnedMesh;
}

//sets up a part of the ragdoll
//int index = the index number of the part
//int setMeshBoneTransformIndex = the bone index that this part is
linked to,
//float offsetX, float offsetY, float offsetZ = translatin offset for
the part in bone local space
//float mass = part's mass,
//btCollisionShape * a_shape = part's collision shape
void RagDoll::SetPart(int index, int setMeshBoneTransformIndex, float
offsetX, float offsetY, float offsetZ,float mass, btCollisionShape *
a_shape)
{
  m_boneIndicesToFollow[setMeshBoneTransformIndex] = index;

  //we set the parts position according to the skinned mesh current
position

  D3DXMATRIX t_poseMatrix = m_skinnedMeshContainer->GetPoseMatrix()
[setMeshBoneTransformIndex];
  D3DXMATRIX *t_boneWorldRestMatrix = m_skinnedMesh->GetBoneWorldRestM
atrix(setMeshBoneTransformIndex);

  D3DXMATRIX t_boneWorldPosition;
```

```
  D3DXMatrixMultiply(&t_boneWorldPosition, t_boneWorldRestMatrix, &t_
poseMatrix);

  D3DXVECTOR3 * t_head = m_skinnedMesh->GetBoneHead(setMeshBoneTransf
ormIndex);
  D3DXVECTOR3 * t_tail = m_skinnedMesh->GetBoneTail(setMeshBoneTransf
ormIndex);

  float tx = t_tail->x - t_head->x;
  float ty = t_tail->y - t_head->y;
  float tz = t_tail->z - t_head->z;

  //part's world matrix
  D3DXMATRIX *t_partMatrix = new D3DXMATRIX();
  *t_partMatrix = t_boneWorldPosition;

  D3DXMATRIX *t_centerOffset = new D3DXMATRIX();
  D3DXMatrixIdentity(t_centerOffset);
  D3DXMatrixTranslation(t_centerOffset, (tx / 2.0f) + offsetX, (ty /
2.0f) + offsetY, (tz/2.0f) + offsetZ);
  D3DXMatrixMultiply(t_partMatrix, t_partMatrix, t_centerOffset);

  D3DXVECTOR3 t_pos;
  D3DXVECTOR3 t_scale;
  D3DXQUATERNION t_rot;

  D3DXMatrixDecompose(&t_scale, &t_rot, &t_pos, t_partMatrix);

  btRigidBody* body = PhysicsFactory::GetInstance()-
>CreateRigidBody(mass,t_pos.x, t_pos.y, t_pos.z, t_rot.x, t_rot.y,
t_rot.z, t_rot.w, a_shape);

  D3DXMATRIX t_partInverse;
  D3DXMatrixInverse(&t_partInverse, NULL, t_partMatrix);

  //puts the bone's matrix in part's local space, and store it in m_
boneToPartTransforms
  D3DXMatrixMultiply(m_boneToPartTransforms[setMeshBoneTransformInd
ex], &t_boneWorldPosition, &t_partInverse);

  m_ragdollBodies[index] = body;

  delete t_partMatrix;
```

```
  t_partMatrix = NULL;

  delete t_centerOffset;
  t_centerOffset = NULL;

}

//when a bone is not going to have a part directly linked to it, it 
needs to follow a bone that has
//a part linked to
//int realBoneIndex = the bone that has no part linked
//int followBoneIndex = the bone that has a part linked
void RagDoll::SetBoneRelation(int realBoneIndex, int followBoneIndex)
{
  //it is going to the same thing the setPart method does, but the 
bone it is going to take
  //as a reference is the one passed as followBoneIndex and the the 
part's matrix is below
  //by calling GetPartForBoneIndex. Still there is going to be a new 
entry in m_boneToPartTransforms
  //which is the bone transform in the part's local space
  int partToFollowIndex = GetPartForBoneIndex(followBoneIndex);

  m_boneIndicesToFollow[realBoneIndex] = partToFollowIndex;

  D3DXMATRIX t_poseMatrix = m_skinnedMeshContainer->GetPoseMatrix()
[realBoneIndex];
  D3DXMATRIX *t_boneWorldRestMatrix = m_skinnedMesh->GetBoneWorldRestM
atrix(realBoneIndex);

  D3DXMATRIX t_boneWorldPosition;
  D3DXMatrixMultiply(&t_boneWorldPosition, t_boneWorldRestMatrix, &t_
poseMatrix);

  D3DXMATRIX *t_partMatrix = new D3DXMATRIX();
  btTransform t_partTransform = m_ragdollBodies[partToFollowIndex]-
>getWorldTransform();
  *t_partMatrix = BT2DX_MATRIX(t_partTransform);

  D3DXMATRIX t_partInverse;
  D3DXMatrixInverse(&t_partInverse, NULL, t_partMatrix);

  D3DXMatrixMultiply(m_boneToPartTransforms[realBoneIndex], &t_
boneWorldPosition, &t_partInverse);

  delete t_partMatrix;
```

```
    t_partMatrix = NULL;

  }

  btRigidBody ** RagDoll::GetRadollParts()
  {
    return m_ragdollBodies;
  }

  btTypedConstraint **RagDoll::GetConstraints()
  {
    return m_ragdollConstraints;
  }

  void RagDoll::AddConstraint(btTypedConstraint *a_constraint)
  {
    m_ragdollConstraints[m_constraintCount] = a_constraint;
    m_constraintCount++;
  }

  //This method will return the world position that the given bone
  should have
  D3DXMATRIX * RagDoll::GetBoneWorldTransform(int boneIndex)
  {
    //the part world matrix is fetched, and then we apply the bone
  transform offset to obtain
    //the bone's world position
    int t_partIndex = GetPartForBoneIndex(boneIndex);

    btTransform  t_transform = m_ragdollBodies[t_partIndex]-
  >getWorldTransform();
    D3DXMATRIX t_partMatrix = BT2DX_MATRIX(t_transform);

    D3DXMatrixIdentity(m_bonesCurrentWorldPosition[boneIndex]);
    D3DXMatrixMultiply(m_bonesCurrentWorldPosition[boneIndex], m_boneToP
  artTransforms[boneIndex], &t_partMatrix);

    return m_bonesCurrentWorldPosition[boneIndex];
  }

  int RagDoll::GetPartForBoneIndex(int boneIndex)
  {
    for(int i = 0; i < m_totalBones;i ++)
    {
```

```
        if(i == boneIndex)
        {
          return m_boneIndicesToFollow[i];
        }
      }

      return -1;
    }
```

How it works...

As you can see from the example above, for this example you require a skinned mesh model. The mesh model can either be downloaded from some royalty-free website, or made via Blender or any other 3D software package, such as Maya or Max. As the whole concept of a ragdoll is based on the bones of the mesh, we have to make sure that the 3D model has the bones set up correctly.

After that, there are lots of small parts in the code. The first part of the problem is to write a bone container class, which stores all the bone information. Next, we need to use the bone container class and by using the Bullet physics SDK, assign a rigid body to each of the bones. After the rigid body has been set up, we need to traverse through the bones once again and create a relationship between each bone, so that when one bone moves, the neighboring bones move as well. Finally, we also need to add constraints so that when the physics engine simulates the ragdoll, it can solve the constraints properly and output the result to the bones.

10
Multithreading in Game Development

In this chapter, the following recipes will be covered:

- Concurrency in games – creating a thread
- Joining and detaching a thread
- Passing arguments to a thread
- Avoiding deadlocks
- Data race and mutex
- Writing a thread-safe class

Introduction

To understand multithreading, let us first understand the meaning of threads. A thread is a concurrent unit of execution. It has its own call stack for methods being invoked, their arguments, and local variables. Each application has at least one thread running when it is started, the main thread. When we talk about multithreading, it means one process has many threads running independently and concurrently, but with shared memory. Often, multithreading is confused with multi-processing. A multiprocessor has multiple processes running, each with its own thread.

Although multithreaded applications may be complex to write, they are lightweight. However, a multithreaded architecture is not well suited for a distributed application. In games, we may have one or more threads running. The golden question is when and why should we use multithreading. Although this is quite subjective, you would use multithreading if you want multiple tasks to happen concurrently. So if you do not want your physics code, or audio code in the game, to wait for the main loop to finish processing, you would multithread the physics and the audio loop.

Concurrency in games – creating a thread

The first step of writing multithreaded code is to spawn a thread. At this point, we must note that the application is already running an active thread, the main thread. So when we spawn a thread, there will be two active threads in the application.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to spawn a thread. Add a source file called `Source.cpp` and add the following code to it:

```
int ThreadOne()
{
  std::cout << "I am thread 1" << std::endl;
  return 0;
}

int main()
{
  std::thread T1(ThreadOne);

  if (T1.joinable()) // Check if can be joined to the main thread
    T1.join();     // Main thread waits for this to finish

  _getch();
  return 0;
}
```

How it works...

The first step is to include the header file, `thread.h`. This gives us access to all the inbuilt libraries that we may need to create our multithreaded application. The next step is to create the task or the function that we need to thread. In this example, we have created a function called `ThreadOne`. This function represents any function that we can use to multithread. This could be a physics function, or audio, or anything that we may desire. For simplicity, we have used a function that prints a message. The next step is to spawn a thread. We simply need to write the keyword `thread`, assign a name to the thread (`T1`) and then write the function/task that we want to thread. In this case, it is `ThreadOne`.

This spawns a thread and will not execute independently of the main thread.

Joining and detaching a thread

After a thread is spawned, it starts executing as a new task, separate from the main thread. However, there may be situations in which we want the task to rejoin the main thread. This is possible. We may also want that the thread always stays apart from the main thread. That is also possible. However, there are a few precautions that we must take when attaching to and detaching from the main thread.

Getting ready

You need to have a working Windows machine and Visual Studio.

How to do it...

In this recipe we will see how easy it is to join and detach threads. Add a source file called `Source.cpp`. Add the following code to it:

```
int ThreadOne()
{
  std::cout << "I am thread 1" << std::endl;
  return 0;
}

int ThreadTwo()
{
  std::cout << "I am thread 2" << std::endl;
  return 0;
}

int main()
```

```
{
  std::thread T1(ThreadOne);
  std::thread T2(ThreadTwo);

  if (T1.joinable()) // Check if can be joined to the main thread
    T1.join();     // Main thread waits for this to finish


  T2.detach();    //Detached from main thread

  _getch();
  return 0;
}
```

How it works...

In the preceding example, at first two threads are spawned. The two threads are `T1` and `T2`. When the threads are spawned, they act independently and concurrently. However, when there is a need for any thread to be joined back to the main thread, we can do that as well. First, we need to check whether the thread can be joined to the main thread. We accomplish this with the joinable function. If the function returns `true`, the thread can join to the main thread. We can join to the main thread with the `join` function. If we directly join, without first checking whether the thread can join to the main thread, it may cause issues with the main thread failing to accept the thread. After the thread joins to the main thread, the main thread waits for that thread to finish.

If we want to detach a thread from the main thread, we can use the `detach` function. However, after we detach it from the main thread, it is detached forever.

Passing arguments to a thread

Like in functions, we may also want to send parameters and arguments to the thread. As threads are just tasks, and tasks are just a collection of functions, it is necessary to understand how to send arguments to a thread. If we can send arguments to a thread at runtime, then the thread can perform all the operations dynamically. In most cases, we would thread the physics, AI, or audio sections of the code. All these sections would require functions that take in arguments.

Getting ready

You need a Windows machine and a working copy of Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to add a heuristic function to our game for pathfinding. Add a source file called `Source.cpp`. Add the following code to it:

```
class Wrapper
{
public:
  void operator()(std::string& msg)
  {
    msg = " I am from T1";
    std::cout << "T1 thread initiated" << msg << std::endl;


  }
};

int main()
{
  std::string s = "This is a message";
  std::cout << std::this_thread::get_id() << std::endl;

  std::thread T1((Wrapper()), std::move(s));
  std::cout << T1.get_id() << std::endl;

  std::thread T2 = std::move(T1);
  T2.join();



  _getch();

}
```

How it works...

The best way to do pass arguments is to write a `Wrapper` class and overload the `()` operator. After we overload the `()` operator, we can send arguments to the thread. To do this, we create a string and store the string in a variable. Then we need to spawn a thread as usual; however, instead of just passing in the function name, we pass in the class name and the string. In threads, we need to pass the arguments by reference, so we could use the `ref` function. However, a better way to do this is by using the `move` function, where we note the memory location itself and pass it to the argument. The `operator` function accepts the string and prints the message.

If we want to create a new thread and make it the same as the first thread, we can again use the `move` function to do this. In addition to this, we can get the thread's ID by using the `get_id` function.

Avoiding deadlocks

When two or more tasks want to use the same resource, we have a race condition. Until one task finishes using the resource, the other task cannot get access to it. This is known as a **deadlock**, and we must avoid deadlocks at all costs. For example, resource `Collision` and resource `Audio` are used by process `Locomotion` and process `Bullet`:

- `Locomotion` starts to use `Collision`
- `Locomotion` and `Bullet` try to start using `Audio`
- `Bullet` "wins" and gets `Audio` first
- Now `Bullet` needs to use `Collision`
- `Collision` is locked by `Locomotion`, which is waiting for `Bullet`

Getting ready

For this recipe, you will need a Windows machine and an installed copy of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to avoid deadlocks:

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

using namespace std;

void Physics()
{
  for (int i = 0; i > -100; i--)
    cout << "From Thread 1: " << i << endl;

}

int main()
{
  std::thread t1(Physics);
  for (int i = 0; i < 100; i++)
```

```
    cout << "From main: " << i << endl;

  t1.join();

  int a;
  cin >> a;
  return 0;
}
```

How it works...

In the preceding example, we have spawned a thread, `t1`, which starts a function to print numbers from 0 to -100, decreasing by 1. There is also a main thread, which starts to print numbers from 0 to 100, increasing by 1. Again, we have chosen these functions for simplicity of understanding. These could easily be replaced by an *A** algorithm and a search algorithm, or anything we want.

If we look at the console output, we notice that it is quite messy. The reason for that is the `cout` object is being used by both the main thread and `t1`. Hence there is a data race condition taking place. Whoever is winning at each turn is getting to display the number. We must avoid this kind of programming structure at all costs. More often than not, it will cause a deadlock and disruption.

Data race and mutex

Data race conditions are very common in multithreaded applications, but we must avoid such a scenario so that deadlocks do not happen. A **mutex** helps us to overcome deadlocks. A mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio.

How to do it...

In this recipe, we will see how easy it is to understand data races and mutexes. Add a source file called `Source.cpp` and add the following code to it:

```
#include <thread>
#include <string>
#include <mutex>
```

```
#include <iostream>

using namespace std;

std::mutex MU;

void Locomotion(string msg, int id)
{
  std::lock_guard<std::mutex> guard(MU); //RAII
  //MU.lock();
  cout << msg << id << endl;
  //MU.unlock();
}
void InterfaceFunction()
{
  for (int i = 0; i > -100; i--)
    Locomotion(string("From Thread 1: "), i);

}

int main()
{
  std::thread FirstThread(InterfaceFunction);
  for (int i = 0; i < 100; i++)
    Locomotion(string("From Main: "), i);

  FirstThread.join();

  int a;
  cin >> a;
  return 0;
}
```

How it works...

In this example, both the main thread and `t1` want to display some numbers. However, as both of them want to use the `cout` object, it creates a data race situation. To avoid this, one approach is to use mutex locks. So before executing the `print` statement, we have `mutex.lock`, and after the `print` statement, we have `mutex.unlock`. This will work, and prevent the data race condition, as mutex will allow one thread to use the resource and make the other thread wait for it. However, this program is not yet thread safe. This is because if the `cout` statement throws an error or exception, the mutex will never get unlocked and the other threads will always be in a `wait` state.

To prevent this, we will use the **Resource Acquisition is Initialization technique** (**RAII**) of C++. We add an inbuilt lock guard to the function. This code is exception-safe because C++ guarantees that all stack objects are destroyed at the end of the enclosing scope, known as **stack unwinding**. The destructors of both the lock and file objects are therefore guaranteed to be called when returning from the function, whether an exception has been thrown or not. Therefore, it will not stop other threads from waiting eternally if an exception has occurred. Despite doing this, this application is not thread safe. This is because the `cout` object is a global object, so other parts of the program can access it as well. Therefore, we need to encapsulate this even further. This we will see later.

Writing a thread-safe class

When dealing with multiple threads, writing a thread-safe class becomes an absolute must. If we do not write classes that are thread safe, there are many complications that may arise, such as deadlocks. We must also keep in mind that when we write a thread-safe class, there is no potential danger from data races and mutex.

Getting ready

For this recipe, you will need a Windows machine and an installed version Visual Studio.

How to do it...

In this recipe, we will see how easy it is to write a thread safe class in C++. Add a source file called `Source.cpp` and add the following code to it:

```
#include <thread>
#include <string>
#include <mutex>
#include <iostream>
#include <fstream>

using namespace std;

class DebugLogger
{
  std::mutex MU;
  ofstream f;
public:
  DebugLogger()
  {
    f.open("log.txt");
  }
```

```
    void ResourceSharingFunction(string id, int value)
    {
      std::lock_guard<std::mutex> guard(MU); //RAII
      f << "From" << id << ":" << value << endl;
    }

  };



  void InterfaceFunction(DebugLogger& log)
  {
    for (int i = 0; i > -100; i--)
      log.ResourceSharingFunction(string("Thread 1: "), i);

  }

  int main()
  {
    DebugLogger log;
    std::thread FirstThread(InterfaceFunction,std::ref(log));
    for (int i = 0; i < 100; i++)
      log.ResourceSharingFunction(string("Main: "), i);

    FirstThread.join();

    int a;
    cin >> a;
    return 0;
  }
```

How it works...

In the previous recipe, we saw how, despite writing a mutex and locks, our code was not thread safe. This is because we were using a global object, `cout`, which could be accessed from other parts of the code as well and so was not thread safe. So we have avoided doing this by adding one more layer of abstraction and outputting the result to a log file.

We have created a class called `Logfile`. Inside this class, we have created a lock guard and a mutex. On top of that, we have also created a stream object called `f`. Using this, we output the contents to a text file. The threads that need access to this functionality will need to create an object of the `LogFile` and then use the function appropriately. We are using the lock guard in the RAII system. Because of this layer of abstraction, there is no chance that the functionality can be used externally and it is quite safe.

However, even in this program, we need to take certain precautions. The first precaution that we should take is that we should not return `f` from any function. Also, we have to be careful that `f` should not be directly available from any other class or external function. If we do either of the above, the resource `f` will again be available to external sections of the program, will not be protected, and will therefore no longer be thread safe.

11
Networking in Game Development

In this chapter, the following recipes will be covered:

- Understanding the different layers
- Selecting the appropriate protocol
- Serializing the packets
- Using socket programming in games
- Sending the data
- Receiving the data
- Dealing with lag
- Using synchronized simulation
- Using area of interest filtering
- Using local perception filtering

Introduction

In the modern era of video gaming, networking plays a huge role in the overall playability of a game. A single player game offers an average of about 15-20 hours of gameplay. However, with the multiplayer (networked) feature, the gameplay time increases exponentially, as now the users have to play against other human opponents and improve their tactics. Whether it is a PC game, console or mobile, having multiplayer capabilities has become a common feature these days. From a freemium model for games, where the monetization and revenue model is based around in-app purchases and ads, it is necessary for the game to have thousands or millions or active users per day. That is the only way the game will make money. When we speak about multiplayer, we should not fool ourselves by thinking that this is restricted to **PvP** (**player versus player**) in real time. It can also be asynchronous multiplayer, where the player competes with the *data* from an active player's deck but not with the player themselves. It gives the illusion that the player is competing against a real player. Also, with the advent of social media, networking also plays a role in helping you compete against your friends. For example, in *Candy Crush*, after you finish a level, you are shown how your friends fared in the same level and who the next friend to beat is. All this adds to the hype around the game and compels you to keep playing it.

Understanding the different layers

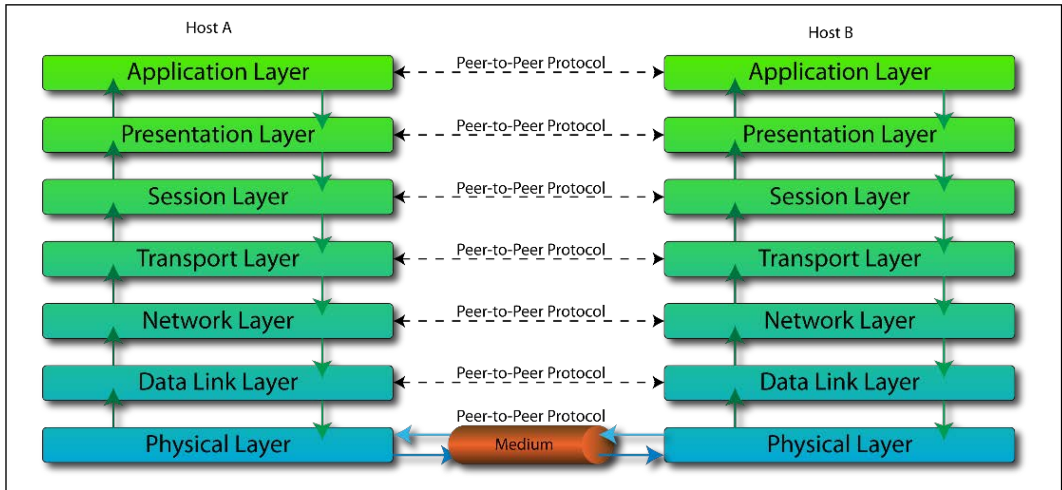
From a technical point of view, the entire networking model is divided into multiple layers. This model is also referred to as the **OSI** (**Open Systems Interconnection**) model. Each layer has a special significance and must be understood properly to be able to interact with other layers of the topology.

Getting ready

To work through this recipe, you will need a machine running Windows.

How to do it...

In this recipe, we will see how easy it is to understand the different layers of the networking topology. Look at the following diagram:



How it works...

To understand the OSI model, we have to look at the model from the bottom of the stack to the top. The layers of the OSI model are:

- **Physical layer**: This establishes the actual physical connection to the network. This is defined by whether we are using copper wire or fiber optics. It defines the network topology that is used, ring or bus, and so on. It also defines the transmission mode: whether it is simplex, half duplex, or full duplex.
- **Data link layer**: This provides the actual link between two connected nodes. The data link layer has two sublayers: the **MAC** layer (**Media Access Control**) and the **LLC** layer (**Logical Link Control**).
- **Network layer**: This layers provides the functional means of transfer of variable length data called **datagrams**. The transfer happens from one connected node to another on the same network. This forms the IP.

- **Transport layer**: This layer also provides the functional means of transferring data. The data is transferred from a source to a destination, travelling via one or more networks. Some of the protocols used here are TCP and UDP. **TCP** is the **transfer control protocol** and is a secured connection. **UDP** is the **user datagram protocol** and is the less secure one. In video games, we use both TCP and UDP protocols. When there is a situation where the user has to log in to the server, we use TCP as it is more secure, because the next information from the client is not sent unless there is an acknowledgement from the server concerning the previous data. It can be slow, however, so if security is more important than speed, we use TCP. After the user logs in, the game starts after other players have joined. Now we use UDP for the majority of situations, as speed is more important than security and a few dropped packets could have a huge impact. UDP packets are not always received as there is no acknowledgement.
- **Session layer**: This layer controls the connections between the network and the remote computer. This layer is responsible for establishing, managing, and terminating a connection.
- **Presentation layer**: This layer controls the different semantics that need to be established between the connections. All the encryption logic is written in this layer.
- **Application layer**: This layer deals with the communication with the software application itself. This is the closest layer from the end user's point of view.

Selecting the appropriate protocol

In games, most of the time there is an important decision that must be made: whether to use TCP or UDP. The decision often ends up in favor of UDP, but still it is important to understand the difference between the two.

Getting ready

You need a Windows machine. No other prerequisites are required.

How to do it…

In this recipe, we will find out how easy it is to make a decision on whether to use TCP or UDP.

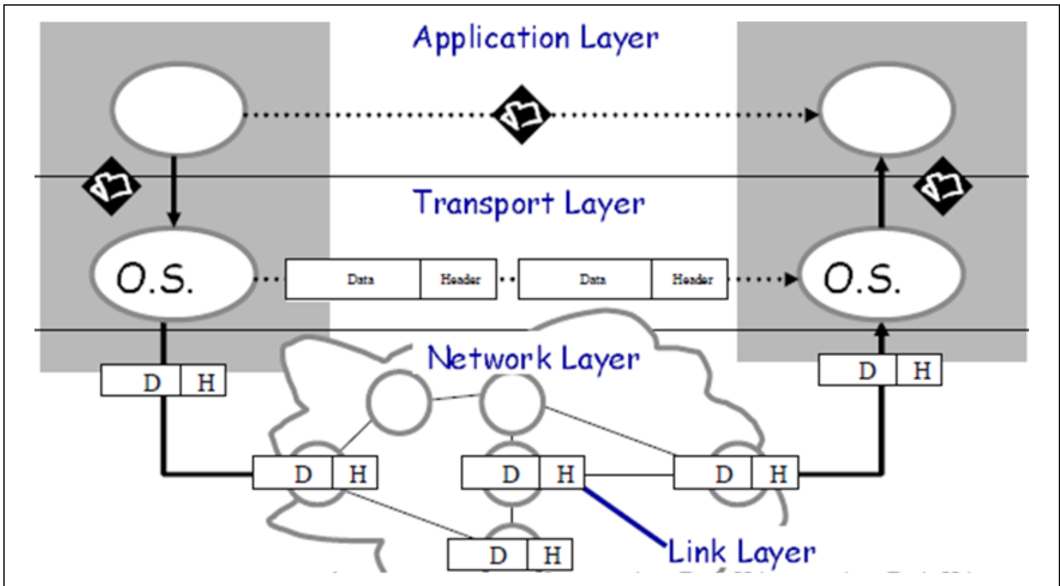
Ask the following questions:

- Does the system require guaranteed delivery?
- Is there a requirement for retransmission?

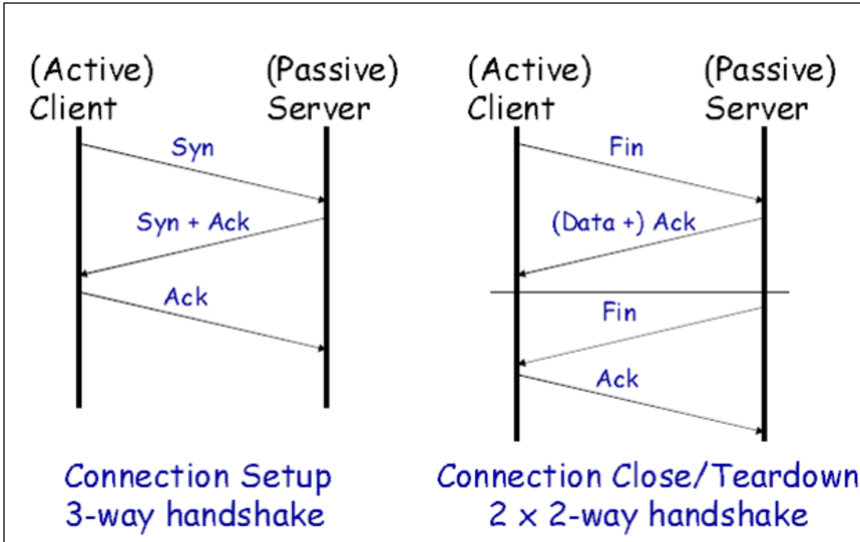
- Does the system require any handshaking mechanism?
- What kind of congestion control does it need?
- Is speed a consideration in the system?

How it works...

TCP and UDP are built on top of the IP layer:



A TCP connection is considered reliable because there is a two-way handshake system enabled. Once the message is delivered to the endpoint, an acknowledgement message is sent out. It supports various other services as well, such as congestion control and multiplexing. The fact that TCP is also full duplex makes it quite a potent connection to use. The way it handles the reliable transfer of data is through byte sequence numbers. It sets a timeout function and, based on timeouts, it can decide whether a package has been delivered or not. The following diagram shows how the handshaking protocol is established:



Another mechanism for TCP is the sliding window mechanism, which guarantees the reliable delivery of data. It ensures that the data packets are delivered in a sequential manner and a flow control between the sender and receiver is established.

UDP is used when we are not too concerned about the data packets being delivered out of order. The main concern is how fast the packets are delivered. There is no reliability and no guarantee that the packets will be delivered.

Applications that require ordered delivery must restore datagram ordering themselves. Datagrams can be written to a target address without knowing if it exists or is listening. Messages can also be broadcast to all hosts on a particular subnet. *DOOM* did this. Sometimes, if we require minimal reliability, UDP is open to adding that functionality. At that point it is also referred to as reliable UDP.

Serializing the packets

Serialization is a key feature to have in a networking system. The process of serialization involves converting a message or data to a format that can be transmitted over the network, and then decoding it. There are a variety of ways to serialize and deserialize data, and it comes down to a matter of personal choice.

Getting ready

You need to have a working Windows machine and Visual Studio. No other requirements are needed.

How to do it…

In this recipe, we will see how easy it is to serialize data. Create a source file and derive it from the serializer class:

```
using namespace xmls;

class LastUsedDocument: public Serializable
{
public:
  LastUsedDocument();
  xString Name;
  xString Path;
  xInt Size;
};

class DatabaseLogin: public Serializable
{
public:
  DatabaseLogin();
  xString HostName;
  xInt Port;
  xString User;
  xString Password;
};

class SerialisationData: public Serializable
{
```

```
public:
  SerialisationData();
  xString Data1;
  xString Data2;
  xString Data3;
  xInt Data4;
  xInt Data5;
  xBool Data6;
  xBool Data7;
  DatabaseLogin Login;
  Collection<LastUsedDocument> LastUsedDocuments;
};

LastUsedDocument::LastUsedDocument()
{
  setClassName("LastUsedDocument");
  Register("Name", &Name);
  Register("Path", &Path);
  Register("Size", &Size);
};

DatabaseLogin::DatabaseLogin()
{
  setClassName("DatabaseLogin");
  Register("HostName", &HostName);
  Register("Port", &Port);
  Register("User", &User);
  Register("Password", &Password);
};

SerialisationData::SerialisationData()
{
  setClassName("SerialisationData");
  Register("Data1", &Data1);
  Register("Data2", &Data2);
  Register("Data3", &Data3);
  Register("Data4", &Data4);
  Register("Data5", &Data5);
  Register("Data6", &Data6);
  Register("Data7", &Data7);
```

```
  Register("Login", &Login);
  Register("LastUsedDocuments", &LastUsedDocuments);
  setVersion("2.1");
};

int main()
{
  // Creating the Datas object
  cout << "Creating object..." << endl;
  SerialisationData *Datas=new SerialisationData;
  Datas->Data1="This is the first string";
  Datas->Data2="This is the second random data";
  Datas->Data3="3rd data";
  Datas->Data4=1234;
  Datas->Data5=5678;
  Datas->Data6=false;
  Datas->Data7=true;
  Datas->Login.HostName="aws.localserver.something";
  Datas->Login.Port=2000;
  Datas->Login.User="packt.pub";
  Datas->Login.Password="PacktPassword";

  for (int docNum=1; docNum<=10; docNum++)
  {
    LastUsedDocument *doc = Datas->LastUsedDocuments.newElement();
    std::stringstream docName;
    docName << "Document #" << docNum;
    doc->Name = docName.str();
    doc->Path = "{FILEPATH}"; // Set Placeholder for search/replace
    doc->setVersion("1.1");
  }

  cout << "OK" << endl;

  // Serialize the Datas object
  cout << "Serializing object... " << endl;
  string xmlData = Datas->toXML();
  cout << "OK" << endl << endl;
  cout << "Result:" << endl;
  cout << xmlData << endl << endl;

  cout << "Login, URL:" << endl;
```

```
    cout << "Hostname: " << Datas->Login.HostName.value();
    cout << ":" << Datas->Login.Port.toString() << endl << endl;
    cout << "Show all collection items" << endl;
    for (size_t i=0; i<Datas->LastUsedDocuments.size(); i++)
    {
      LastUsedDocument* doc = Datas->LastUsedDocuments.getItem(i);
      cout << "Item " << i << ": " << doc->Name.value() << endl;
    }
    cout << endl;

    cout << "Deserialization:" << endl;
    cout << "Class version: " << Serializable::IdentifyClassVersion(xml
  Data) << endl;
    cout << "Performing deserialization..." << endl;

    // Deserialize the XML text
    SerialisationData* dser_Datas=new SerialisationData;
    if (Serializable::fromXML(xmlData, dser_Datas))
    {
      cout << "OK" << endl << endl;

      // compare both objects
      cout << "Compareing objects: ";
      if (dser_Datas->Compare(Datas))
        cout << "equal" << endl << endl;
  else
        cout << "net equal" << endl << endl;

      // now set value
      cout << "Set new value for field >password<..." << endl;
      dser_Datas->Login.Password = "newPassword";
      cout << "OK" << endl << endl;

      cout << "compare objects again: ";
      if (dser_Datas->Compare(Datas))
        cout << "equal" << endl << endl; else
        cout << "net equal" << endl << endl;


      cout << "search and replace placeholders: ";
      dser_Datas->Replace("{FILEPATH}", "c:\\temp\\");
```

```
        cout << "OK" << endl << endl;

        //output xml-data
        cout << "Serialize and output xml data: " << endl;
        cout << dser_Datas->toXML() << endl << endl;

        cout << "Clone object:" << endl;
        SerialisationData *clone1(new SerialisationData);
        Serializable::Clone(dser_Datas, clone1);
        cout << "Serialize and output clone: " << endl;
        cout << clone1->toXML() << endl << endl;
        delete (clone1);
      }
      delete(Datas);
      delete(dser_Datas);
      getchar();
      return 0;
    }
```

How it works...

As mentioned before, to serialize is to convert the data to a format that can be transferred. We can do this by using the Google API, or using the JSON format or YAML. In this example, we have used an XML serializer originally written by Lothar Perr. The original source can be found at `http://www.codeproject.com/Tips/725375/Tiny-XML-Serialization-for-Cplusplus`. The whole idea behind the program is that we convert the data to an XML format. In the class serializable data, we publicly derive it from the serializable class. We create a constructor to register all the data elements and we create the different data elements that we want to be serialized. The data elements are of the type `xString` class. In the constructor, we register each of the data elements. Finally, from the client side, we assign the correct data to be sent and, using the XML serializer class and tinyxml, we generate the required XML. Finally, this XML will be sent across the network and on receipt, it will be decoded using the same logic. XML can sometimes be considered quite heavy and cumbersome for games.

In these situations, it is advisable to use JSON. Some modern engines, such as Unity3D and Unreal Engine, already have an inbuilt JSON parser which could be used to serialize the data. However, XML still continues to be an important format. An example of a possible output from our code is shown here:


```
Clone object:
Serialize and output clone:
<SerializableClass Type="SerialisationData" Version="2.1">
    <Member Name="Setting1">This is the first string</Member>
    <Member Name="Setting2">This is the second random data</Member>
    <Member Name="Setting3">3rd data</Member>
    <Member Name="Setting4">1234</Member>
    <Member Name="Setting5">5678</Member>
    <Member Name="Setting6">false</Member>
    <Member Name="Setting7">true</Member>
    <Class Name="Login" Type="DatabaseLogin" Version="1">
        <Member Name="HostName">aws.localserver.something</Member>
        <Member Name="Port">2000</Member>
        <Member Name="User">packt.pub</Member>
        <Member Name="Password">newPassword</Member>
    </Class>
    <Collection Name="LastUsedDocuments">
        <Class Type="LastUsedDocument" Version="1">
            <Member Name="Name">Document #1</Member>
            <Member Name="Path">c:\temp\</Member>
            <Member Name="Size"></Member>
        </Class>
        <Class Type="LastUsedDocument" Version="1">
            <Member Name="Name">Document #2</Member>
            <Member Name="Path">c:\temp\</Member>
            <Member Name="Size"></Member>
        </Class>
        <Class Type="LastUsedDocument" Version="1">
            <Member Name="Name">Document #3</Member>
            <Member Name="Path">c:\temp\</Member>
            <Member Name="Size"></Member>
        </Class>
        <Class Type="LastUsedDocument" Version="1">
            <Member Name="Name">Document #4</Member>
            <Member Name="Path">c:\temp\</Member>
```


Using socket programming in games

Socket programming is one of the earliest mechanisms for transferring data between end-to-end connections. Even now, if you are comfortable writing socket programming, it is a much better option for a relatively small game than using third party solutions, as they add a lot of extra space.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to write sockets:

```
struct sockaddr_in
{
      short      sin_family;
      u_short      sin_port;
      struct      in_addr sin_addr;
      char      sin_zero[8];
};

int PASCAL connect(SOCKET,const struct sockaddr*,int);
    target.sin_family = AF_INET; // address family Internet
    target.sin_port = htons (PortNo); //Port to connect on
    target.sin_addr.s_addr = inet_addr (IPAddress); //Target IP

    s = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP); //Create socket
    if (s == INVALID_SOCKET)
    {
        return false; //Couldn't create the socket
    }
```

How it works...

When two applications are communicating with each other on different machines, one end of that communication channel is often described as the socket. It is a combination of an IP address and a port. As we use signals or pipes to communicate in an inter-process communication environment on different machines, there is a need for sockets.

Berkley Sockets (**BSD**) was the first internet socket API developed. Developed at the University of Berkley, California, and given freely to all Berkley System distributions of UNIX, it is present on all modern operating systems, which are UNIX variants, including OS X and Linux. Windows Sockets is based on BSD sockets and provides additional functionality to comply with the regular Windows programming model. Winsock2 is the newest API.

Common domains are:

- **AF UNIX**: This address format is UNIX pathname
- **AF INET**: This address format is host and port number

The various protocols can be used in the following ways:

- TCP/IP (virtual circuits): SOCK_STREAM
- UDP (datagram): SOCK_DGRAM

These are the steps to set up a simple socket connection:

1. Create a socket.
2. Bind the socket to an address.
3. Wait for input/output to be ready on the socket.
4. Read and write to/from the socket.
5. Repeat from step 3 until you are done.
6. Close the socket.

These steps are explained here with examples:

- `int socket(domain, type, protocol):`

  The parameter `domain` should be set to `PF_INET` (protocol family) and the `type` is the connection type that it should be using. Use `SOCK_STREAM` for a byte stream socket, whereas `SOCK_DGRAM` is used for a datagram (packet) socket. `protocol` is the Internet protocol that is in use. `SOCK_STREAM` would normally give `IPPROTO_TCP`, and `SOCK_DGRAM` would normally give `IPPROTO_UDP`.

- `int sockfd;`

  ```
  sockfd = socket (PF_INET, SOCK_STREAM, 0):
  ```

  The `socket()` function returns a socket descriptor for use in later system calls or `-1`. When the protocol is set to `0`, the socket chooses the correct protocol based on the type specified.

- `int bind(int Socket, struct sockaddr *myAddress, int AddressLen )`

  The function `bind()` ties the socket to a local address. Socket is the socket descriptor. `myAddress` is the local IP address and port. The `AddressSize` parameter gives the size (in bytes) of the address and `bind()` returns `-1` on error.

- `struct sockaddr_in {`

  ```
    short int sin_family;     // set to AF_INET
    unsigned short int sin_port;   // Port number
    struct in_addr sin_addr;   // Internet address
    unsigned char sin_zero[8];   //set to all zeros
  }
  ```

  `struct sockaddr_in` is a parallel structure which makes it easy to reference elements of the socket address. `sin_port` and `sin_addr` must be in Network Byte Order.

Sending the data

After we have correctly set up the sockets, the next step is to create the correct server and client architecture. Sending data is pretty simple and just involves a few lines of code.

Getting ready

To work through this recipe, you will need a machine running Windows with Visual Studio installed.

How to do it...

In this recipe, we will see how easy it is to send data:

```
// Using the SendTo Function
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>
#include <conio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main()
{

  int iResult;
  WSADATA wsaData;

  SOCKET SenderSocket = INVALID_SOCKET;
  sockaddr_in ReceiverAddress;

  unsigned short Port = 27015;

  char SendBuf[1024];
```

```
    int BufLen = 1024;

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
      wprintf(L"WSAStartup failed with error: %d\n", iResult);
      return 1;


    }

    //-----------------------------------------------
    // Create a socket for sending data
    SenderSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (SenderSocket == INVALID_SOCKET) {
      wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
      WSACleanup();
      return 1;
    }
    //-----------------------------------------------
    // Set up the ReceiverAddress structure with the IP address of
    // the receiver (in this example case "192.168.1.1")
    // and the specified port number.
    ReceiverAddress.sin_family = AF_INET;
    ReceiverAddress.sin_port = htons(Port);
    ReceiverAddress.sin_addr.s_addr = inet_addr("192.168.1.1");

    //-----------------------------------------------
    // Send a datagram to the receiver
    wprintf(L"Sending a datagram to the receiver...\n");
    iResult = sendto(SenderSocket,
      SendBuf, BufLen, 0, (SOCKADDR *)& ReceiverAddress,
  sizeof(ReceiverAddress));
    if (iResult == SOCKET_ERROR) {
      wprintf(L"sendto failed with error: %d\n", WSAGetLastError());
      closesocket(SenderSocket);
      WSACleanup();
      return 1;
    }
    //-----------------------------------------------
    // When the application is finished sending, close the socket.
    wprintf(L"Finished sending. Closing socket.\n");
    iResult = closesocket(SenderSocket);
```

```
  if (iResult == SOCKET_ERROR) {
    wprintf(L"closesocket failed with error: %d\n",
WSAGetLastError());
    WSACleanup();
    return 1;
  }
  //----------------------------------------------
  // Clean up and quit.
  wprintf(L"Exiting.\n");
  WSACleanup();

  getch();
  return 0;
}

//Using the Send Function
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT 27015

int main() {

  //---------------------
  // Declare and initialize variables.
  int iResult;
  WSADATA wsaData;

  SOCKET ConnectSocket = INVALID_SOCKET;
  struct sockaddr_in clientService;

  int recvbuflen = DEFAULT_BUFLEN;
```

```
    char *sendbuf = "Client: sending data test";
    char recvbuf[DEFAULT_BUFLEN] = "";

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
      wprintf(L"WSAStartup failed with error: %d\n", iResult);
      return 1;
    }

    //----------------------
    // Create a SOCKET for connecting to server
    ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ConnectSocket == INVALID_SOCKET) {
      wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
      WSACleanup();
      return 1;
    }

    //----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port of the server to be connected to.
    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr("127.0.0.1");
    clientService.sin_port = htons(DEFAULT_PORT);

    //----------------------
    // Connect to server.
    iResult = connect(ConnectSocket, (SOCKADDR*)&clientService,
  sizeof(clientService));
    if (iResult == SOCKET_ERROR) {
      wprintf(L"connect failed with error: %d\n", WSAGetLastError());
      closesocket(ConnectSocket);
      WSACleanup();
      return 1;
    }

    //----------------------
    // Send an initial buffer
    iResult = send(ConnectSocket, sendbuf, (int)strlen(sendbuf), 0);
    if (iResult == SOCKET_ERROR) {
      wprintf(L"send failed with error: %d\n", WSAGetLastError());
      closesocket(ConnectSocket);
```

```
      WSACleanup();
      return 1;
    }

    printf("Bytes Sent: %d\n", iResult);

    // shutdown the connection since no more data will be sent
    iResult = shutdown(ConnectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
      wprintf(L"shutdown failed with error: %d\n", WSAGetLastError());
      closesocket(ConnectSocket);
      WSACleanup();
      return 1;
    }

    // Receive until the peer closes the connection
    do {

      iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
      if (iResult > 0)
        wprintf(L"Bytes received: %d\n", iResult);
      else if (iResult == 0)
        wprintf(L"Connection closed\n");
      else
        wprintf(L"recv failed with error: %d\n", WSAGetLastError());

    } while (iResult > 0);


    // close the socket
    iResult = closesocket(ConnectSocket);
    if (iResult == SOCKET_ERROR) {
      wprintf(L"close failed with error: %d\n", WSAGetLastError());
      WSACleanup();
      return 1;
    }

    WSACleanup();
    return 0;
  }
```

How it works...

The function that is used to communicate over the network is called `sendto`. This is declared as `int sendto (int sockfd, const void *msg, int len, int flags);`.

`sockfd` is the socket descriptor you want to send data to (returned by `socket()` or got from `accept()`), whereas `msg` is a pointer to the data you want to send. `len` is the length of that data in bytes. For simplicity, we can set that `flag` to `0` for now. `sendto()` returns the number of bytes actually sent (it may be less than the number you told it to send) or `-1` on error. By using just this function, you are able to send messages or data from one connection point to the other. This function can be used to send data across the network using the inbuilt Winsock functionality. The `send` function is used for streams of data and hence used for TCP. If we are to use datagrams and connectionless protocols, then we need to use the `sendto` function.

Receiving the data

After we have correctly set up the sockets and sent the data, the next step is to receive the data. Receiving data is pretty simple and just involves a few lines of code.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio.

How to do it...

In this recipe, we will see how easy it is to receive data over the network. There are two ways to do it, either by using the `recv` function or by using the `recvfrom` function:

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

int __cdecl main() {

  //----------------------
```

```
  // Declare and initialize variables.
  WSADATA wsaData;
  int iResult;

  SOCKET ConnectSocket = INVALID_SOCKET;
  struct sockaddr_in clientService;

  char *sendbuf = "this is a test";
  char recvbuf[DEFAULT_BUFLEN];
  int recvbuflen = DEFAULT_BUFLEN;

  //----------------------
  // Initialize Winsock
  iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
  if (iResult != NO_ERROR) {
    printf("WSAStartup failed: %d\n", iResult);
    return 1;
  }

  //----------------------
  // Create a SOCKET for connecting to server
  ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (ConnectSocket == INVALID_SOCKET) {
    printf("Error at socket(): %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
  }

  //----------------------
  // The sockaddr_in structure specifies the address family,
  // IP address, and port of the server to be connected to.
  clientService.sin_family = AF_INET;
  clientService.sin_addr.s_addr = inet_addr("127.0.0.1");
  clientService.sin_port = htons(27015);

  //----------------------
  // Connect to server.
  iResult = connect(ConnectSocket, (SOCKADDR*)&clientService,
sizeof(clientService));
  if (iResult == SOCKET_ERROR) {
    closesocket(ConnectSocket);
    printf("Unable to connect to server: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
```

```
    }

    // Send an initial buffer
    iResult = send(ConnectSocket, sendbuf, (int)strlen(sendbuf), 0);
    if (iResult == SOCKET_ERROR) {
      printf("send failed: %d\n", WSAGetLastError());
      closesocket(ConnectSocket);
      WSACleanup();
      return 1;
    }

    printf("Bytes Sent: %ld\n", iResult);

    // shutdown the connection since no more data will be sent
    iResult = shutdown(ConnectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
      printf("shutdown failed: %d\n", WSAGetLastError());
      closesocket(ConnectSocket);
      WSACleanup();
      return 1;
    }

    // Receive until the peer closes the connection
    do {

      iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
      if (iResult > 0)
        printf("Bytes received: %d\n", iResult);
      else if (iResult == 0)
        printf("Connection closed\n");
      else
        printf("recv failed: %d\n", WSAGetLastError());

    } while (iResult > 0);

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
  }
```

How it works...

Just like the `send` function, only one function is used to receive the data over the network, which can be declared as follows:

```
int recv(int sockfd, void *buf,  int len, int flags);
```

`sockfd` is the socket descriptor to read from. The next parameter, `buf`, is the buffer to read the information into, whereas `len` is the maximum length of the buffer. The next parameter, `recv()`, returns the number of bytes actually read into the buffer or `-1` on error. If `recv()` returns `0`, the remote side has closed the connection on you.

Using this line of code, we can receive data over the network. If the data is serialized while sending, we have to then take the data and deserialize the data at this point. This process will vary based on the method used to serialize the data.

Dealing with lag

One of the major problems that occurs in a networked game is latency or lag. When two players are playing against each other, and one is on a high-speed network and the other is on a very low-speed network, how do we update the data? We need to update it in such a way that it looks normal to both players. No player should get an undue advantage because of this situation.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio.

How to do it...

In this recipe, you will see a few techniques for countering lag.

Generally, a networked game will have the following update loop. We need to figure out, from the loop structure, what is the best way to counter lag:

```
read_network_messages()
    read_local_input()
    update_world()
    send_network_updates()
    render_world()
```

How it works...

In most computer games, when networking is implemented, a specific type of client-server architecture is chosen. Often, an authoritative server is chosen. This means the server dictates the time, results, and other factors. The client is basically *dumb* and all it does is a simulation based on data from the server. Now let us consider that two players are playing a multiplayer FPS game. One of them is on a high-speed connection and the other connection is very slow. So, if the client is dependent on the server for its updates, it will be very difficult to accurately render the positions of the players on the client side. Let's say `UserA` is on a high-speed connection whereas `UserB` is on a low-speed one. `UserA` fires a bullet at `UserB`. Note `UserA` and `UserB` are also moving in the world space. How do we calculate the position of the bullet and the position of each individual player? If we render exactly the information that is coming from the server, it will not be accurate as `UserA` would have already moved to a new position by the time `UserB` gets an update. To counter this, there are two commonly used solution. One is called client-side prediction. The other method is further divided into two more techniques: interpolation and extrapolation. Note that the round-trip time will be quite acceptable if the computers are connected over LAN. All the problems that are being discussed focus on networking over the Internet.

In client-side prediction, the *dumb* factor is taken out of the client and the client starts predicting, based on previous movement inputs, what the next position and animation states will be. Finally, when it gets an update from the server, the server will correct the mistakes and the position will be transformed to the currently received one. There are loads of problems with this system. If the prediction is wrong, there will be a big jitter as the position is changed to the right one. Also, let us consider sound and VFX effects. If the client at `UserA` predicted that `UserB` was walking and the footsteps sound was played, and later the server informed it that `UserB` was actually in water, how do we suddenly rectify that mistake? The same goes for VFX effects and states. This system was used in a lot of the *Quake* worlds.

The second system has two parts: extrapolation and interpolation. In extrapolation, we render ahead of time. This is in some way similar to prediction. It takes the last known update from the server and then simulates forward in time. Thus, if you are lagging 500 milliseconds behind, and the last update you received was that the other player was running 300 units per second perpendicular to your view, then the client could assume that in *real time* the player has moved 150 units straight ahead from their last known position. The client could then just draw the player at that extrapolated position and the local player could still more or less aim right at the other player. However, the problem with this system is that it will rarely happen like that. The movement of the player may change, the state may change and hence this system should be avoided in most cases.

In interpolation, we always render objects in the past. For instance, if the server is sending 25 updates per second (exactly) of the world state, then we might impose 40 milliseconds of interpolation delay in our rendering. Then, as we render frames, we interpolate the position of the object between the last updated position and the position one update over that 40 milliseconds. Interpolation can be done by using the inbuilt lerp function in C++. As the object gets to the last updated position, we receive a new update from the server (since 25 updates per second means that the updates come in every 40 milliseconds) and we can start moving toward this new position over the next 40 milliseconds. The following picture shows the difference in positions of the hitbox from the server and the client side.

If the packet does not arrive after 40 milliseconds, that is, there is a packet drop, then we have two options. The first option is to extrapolate using the method described above. The other option is to make the player go to an idle state till the next packet is received from the server.

Using synchronized simulation

In a multiplayer game, there may be hundreds or thousands of computers connected at the same time. All of the computers will have different configurations. Speed will vary on all these computers. So the questions is, how do we synchronize the clock over all these systems so that they are all in sync?

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio.

How to do it…

In this recipe, we will look at, from a theoretical perspective, the two ways to synchronize clocks.

Take a look at the following pseudocode:

- Method 1
  1. Send a message to `UserA`. Note the time till he receives the message.
  2. Send a message to `UserB`. Note the time again.
  3. Calculate the median based on the values to decide an update time for updating the clock for both computers.

- Method 2
  1. Let the server do most of the calculations.
  2. Let the client do some local calculations.
  3. When the client receives the update from the server, then either correct its mistakes or interpolate based on the results.

How it works…

When we are trying to synchronize the clock, there are two methods. One method is that the server tries to find a median time to synchronize all the clocks. To do this, we can include the mechanics in the game design itself. The server needs to find out the response time of each client machine, so it has to send out messages. These messages can be to press *R* when ready, or a map is loaded on the client machine and the server takes a note of the time. Finally, when it has got a time from all the machines, it calculates a median and then updates the clock for all the machines at that time. The more messages the server sends out to the machines to calculate this median, the more accurate it will be. However, this in no way guarantees synchronization.

Therefore, a better method is that the server does all the calculations and the client does some local calculations as well, using techniques described in previous recipes. Finally, when the server sends an update to the client, the client can correct itself or interpolate to get the desired result. This is a much better result and a much better system to have.

Using area of interest filtering

When we are writing a networking algorithm, we need to decide on the various objects or states that need updating to or from the server. The higher the number of objects, the more time it will take to serialize and send the data across. Therefore, there is a need to prioritize what needs to be updated every frame and which objects can wait for a few more cycles to be updated.

Getting ready

To work through this recipe, you will need a machine running Windows.

How to do it...

In this recipe, we will see how easy it is to create area of interest filtering:

1. Create a list of all objects in the scene.
2. Add a parameter to each object denoting their priority.
3. Based on that priority number, pass it on to the update logic of the game.

How it works...

In a game, we need to define the objects in a certain priority order. The priority order determines whether they should be updated now or at a later time. The objects that require prioritization depend a lot on the game design and a bit of research. For example, in an FPS game, the objects with high priority would be the person that the user is currently shooting at, the ammunition lying nearby, and of course the enemies in close proximity and their positions. This may be different in the case of an RPG or an RTS, so it definitely varies from one game to another.

After we have tagged each object with a priority number, we can tell the update loop to just use the objects that are priority level 1 and 2 for per-frame updates, and use objects that are priority level 3 and 4 for late updates. This structure can also be modified by creating some sort of priority queue. From the queue, objects are popped out based on different update logic. The lower priority objects are also synced but at a later time, not in the current frame.

Using local perception filter

This is yet another method to combat lag in networked games. This entire concept is mathematically based on the concept of perception. The basis of it is that if objects update and render correctly locally to a player, then we can create an illusion of realism, hence the name local perception filter.

Getting ready

To work through this recipe, you will need a machine running Windows.

How to do it...

In this recipe, we will understand the theoretical concept of how easy it is to implement bullet time. Take a look at the following pseudocode:

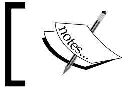
1. Calculate the velocity local to the player.
2. Accelerate the bullet when it starts and slow it down as it reaches the remote player.
3. From the remote player's point of view, the bullet should appear to have been shot at a higher speed than normal speed and then slow down to normal.

How it works...

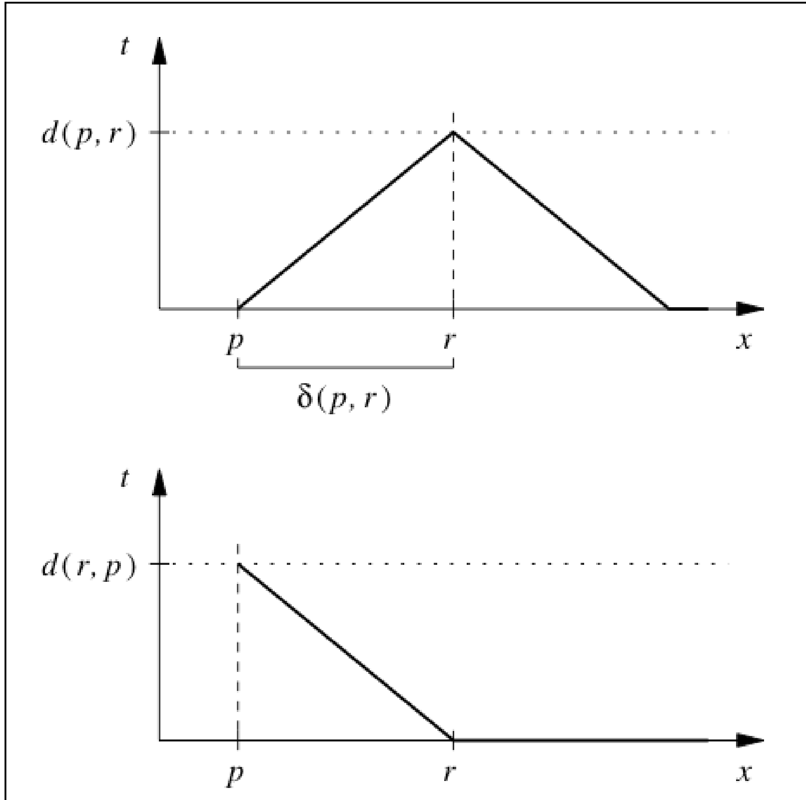
Local perception filters are also called bullet time, and were used for the first time in the movie *The Matrix*. Since then, they have been used in a wide range of games. It is quite easy to do in single player mode; however, in multiplayer it gets a bit more complex as it involves slowing down the rendering. Essentially, the process is to increase and reduce the speed of passive entities when they are near local and remote players. It is a method used to hide communication delays in networked virtual environments and was introduced in *A local perception filter for distributed virtual environments*, *P.M. Sharkey*, (page 242-249). For simplicity, we will call local players *p*, remote players *r*, and passive entities, such as bullets, *e*. Let us say that *d(i,j)* is delay, *delta(i,j)* is distance, and we get the following equations:

$$d(p,e) = \begin{cases} 0, & \text{if } \delta(p,e) = 0, \\ d(p,r), & \text{if } \delta(r,e) = 0. \end{cases}$$

In a graphical format, this can be explained by looking at the following graph. So with respect to *p*, it goes slow uphill and then fast downhill. With respect to *r*, it is faster at the top.

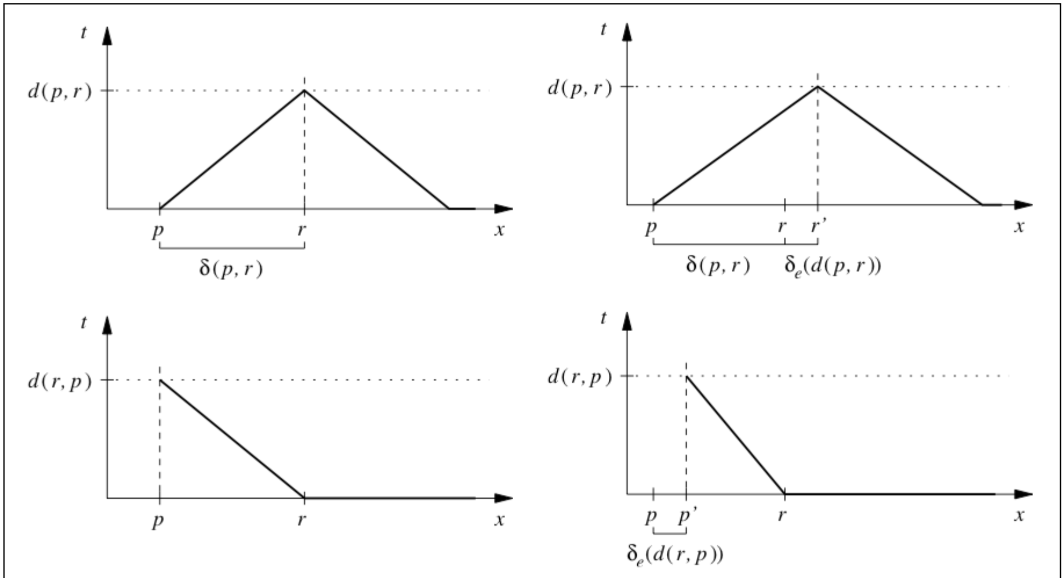


One major limitation of the method is that this cannot be used for *insta-hit* weapons.



The problem is that when *e* reaches r, *p*'s view of e is not there yet, but *e* will speed up anyway in *p*'s view. To tackle this, we introduce a shadow *r*, which buffers *p*'s view of the speedup process.

After adding the buffer, we will get the following revised graphs:



So at the top, won't speed up until *r* is reached, and at the bottom it starts to show *e* at postion *p*. This can also be viewed as a demo at the following URL: `http://mikolalysenko.github.io/local-perception-filter-demo/`.

12
Audio in Game Development

In this chapter, the following recipes are covered:

- Installing FMOD
- Adding background music
- Adding sound effects
- Creating a sound effect manager
- Dealing with multiple sound file names

Introduction

One of the most important aspects in games development is audio programming. However, it is, strangely, one of the most neglected and underrated sections of games development as well. To understand the impact of audio in games, try playing a game such as *Counter-Strike* or *Quake* with sounds and then try playing the games without sound. It has a huge impact. Audio programming, if not done correctly, can lead to games crashing and lots of other problems.

Therefore, it is very important to learn the correct way to do audio programming. Most engines will have a built-in sound component. For others, we need to add the audio component. In this chapter, we will have a look at one of the most popular sound engines. We will also have a look at how to integrate SDL into our C++ code, in order to play audio and sound effects.

Installing FMOD

The first thing to get started with is to install FMOD. This is one of the most popular audio engines and used in almost all modern game engines. It can also be added to any game engine of your choice. The other popular audio engine is called **Wwise**. This is used to integrate the audio for console programming, such as on the PS4.

Getting ready

To work through this recipe, you will need a machine running Windows.

How to do it...

In this recipe, we will see the different types of source control available to us:

1. Go to `http://www.fmod.org/`.
2. To download FMOD, go to `http://www.fmod.org/download/`.

   There is one authoring tool to edit the audio files.However, we should be downloading the FMOD Studio Programmer API and the Low Level Programmer API.

   It also has plugins for all modern engine such as Cocos2d-x, Unreal Engine, and Unity3D.

How it works...

FMOD is a low-level API, so it provides callbacks that help us to use the interface of FMOD to play sounds, pause sounds, and do a whole lot of other things. Because we have the source files, we can build the libraries and also use it in our own engine. FMOD also provides an API for Unity3D, which means that the code is also exposed to C#, making it easier to work with in Unity3D.

Adding background music

Any game would be incomplete if it did not have any background music. So it is very important that we integrate a way to play music into our C++ engine. There are various ways to do this. We are going to use SDL to play music in our game.

Getting ready

You need a Windows machine and a working copy of Visual Studio. The SDL library is also required.

How to do it...

In this recipe, we will find out how easy it is to play background music:

1. Add a source file called `Source.cpp`.
2. Add the following code to it:

```
#include <iostream>
#include "../AudioDataHandler.h"

#include "../lib/SDL2/include/SDL2/SDL.h"

#include "iaudiodevice.hpp"
#include "iaudiocontext.hpp"
#include "audioobject.hpp"

#include "sdl/sdlaudiodevice.hpp"
#include "sdl/sdlaudiocontext.hpp"

#define FILE_PATH "./res/audio/testClip.wav"

int main(int argc, char** argv)
{
  SDL_Init(SDL_INIT_AUDIO);

  IAudioDevice* device = new SDLAudioDevice();
  IAudioContext* context = new SDLAudioContext();

  IAudioData* data = device->CreateAudioFromFile(FILE_PATH);

  SampleInfo info;
  info.volume = 1.0;
  info.pitch = 0.7298149802137;

  AudioObject sound(info, data);
  sound.SetPos(0.0);

  char in = 0;
  while(in != 'q')
  {
    std::cin >> in;
    switch(in)
    {
      case 'a':
```

```
          context->PlayAudio(sound);
          break;
        case 's':
          context->PauseAudio(sound);
          break;
        case 'd':
          context->StopAudio(sound);
          break;
      }
    }

    device->ReleaseAudio(data);
    delete context;
    delete device;

    SDL_Quit();
    return 0;
  }

  int main()
  {
    AudioDataHandler _audioData;
    cout<<_audioData.GetAudio(AudioDataHandler::BACKGROUND);
  }
```

How it works...

In this example, we are playing background music for our game. We need to create an interface as a wrapper to the existing SDL audio library. Interfaces are also good at providing a skeleton architecture that a base class can derive from in the future. We require `SDLAudioDevice`, which is the main handler object for playing the music. On top of that, we create a pointer to an audio data object, which creates audio from the file path provided. The device handler object has a built-in function called `CreateAudioFromFile` to help us with this process. Finally, we have an audio context class, which has functions to play, pause, and stop the audio. Each of the functions takes an audio object as a reference, which is used to perform the action on our audio file.

Adding sound effects

Sound effects are a neat way of adding some sense of tension or achievement to the game. Playing, pausing, and stopping sound effects all work in the same way that we used for background music, which we saw in the previous recipe. However, we can add some variety to the sound files by controlling their position, volume, and pitch.

Getting ready

You need to have a working Windows machine.

How to do it...

Add a source file called `Source.cpp` and add the following code to it:

```
struct SampleInfo
{
  double volume;
  double pitch;
};

SampleInfo info;
info.volume = 1.0;
info.pitch = 0.7298149802137;

AudioObject sound(info, data);
sound.SetPos(0.0);
```

How it works...

In this example, we are only looking at that part of the game that involves modifying the pitch, volume, and position of the sound file. These three things can be considered to be the attributes of the sound file, but there are other attributes as well. Therefore, the first thing to do would be to create a structure. The structure is used to store all the attributes of the sound. All we need to do is populate the structure with values as and when we want them. Finally, we create an audio object and pass in the `SampleInfo` struct as one of the parameters of the object. The constructor then initializes the sound to have these properties. Because we attached the properties to the object, it means we can also manipulate them at runtime and lower the volume dynamically if required. The pitch and other properties could also be manipulated in the same way.

Creating a sound effect manager

Although not one of the best practices out there, one of the most common methods for handling audio is to create a manager class. The manager class should ensure that there is only one audio component in the whole game, which controls which sound is to be played, paused, looped, and so on. Although there are other ways of writing the manager class, this is the most standard practice.

Getting ready

For this recipe, you will need a Windows machine and Visual Studio.

How to do it…

In this recipe, we will find out how easy it is to add sound effect manager using the following snippet:

```
#pragma once
#include <iostream>
#include "../lib/SDL2/include/SDL2/SDL.h"

#include "iaudiodevice.hpp"
#include "iaudiocontext.hpp"
#include "audioobject.hpp"

#include "sdl/sdlaudiodevice.hpp"
#include "sdl/sdlaudiocontext.hpp"

#define FILE_PATH "./res/audio/testClip.wav"

class GlobalAudioClass
{
private:

  AudioObject* _audObj;
  IAudioDevice* device = new SDLAudioDevice();
  IAudioContext* context = new SDLAudioContext();

  IAudioData* data = device->CreateAudioFromFile(FILE_PATH);

  SampleInfo info;


  static GlobalAudioClass *s_instance;

  GlobalAudioClass()
  {
    info.volume = 1.0;
   info.pitch = 0.7298149802137;
    _audObj = new AudioObject(info,data);
  }
```

```
    ~GlobalAudioClass()
    {
      //Delete all the pointers here
    }
  public:
    AudioObject* get_value()
    {
      return _audObj;
    }
    void set_value(AudioObject* obj)
    {
      _audObj = obj;
    }
    static GlobalAudioClass *instance()
    {
      if (!s_instance)
        s_instance = new GlobalAudioClass();
      return s_instance;
    }
  };

  // Allocating and initializing GlobalAudioClass's
  // static data member.  The pointer is being
  // allocated - not the object inself.
  GlobalAudioClass *GlobalAudioClass::s_instance = 0;
```

How it works...

In this example, we have written a singleton class to implement the audio manager. The singleton class has all the necessary `sdl` headers and other device and context objects required to play the sounds. All these are private, so they cannot be accessed from other classes. We also made a static pointer to the class and made the constructor private as well. If we need an instance of this audio manager, we have to use the static `GlobalAudioClass *instance()` function. This function automatically checks whether there is an instance already created, then it returns that instance, or it creates a new one. Hence, only one instance of the manager class exists at all times. We can also use the manager to set and get data for the sound file, for example by setting the path of the sound file.

Dealing with multiple sound file names

In games, there will not be one sound file, but multiple sound files that we will have to deal with. Each will have a different name, type, and location. So it is not a wise move to define all of them separately. It will work, but it will be very messy coding if we have over 20 sound effects in our game, so there is a need for a slight improvement to the code.

Getting ready

For this recipe, you will need a Windows machine and an installed version of an SVN client. A versioned project is also necessary.

How to do it...

In this recipe, you will see how easy it is to deal with multiple sound file names. All you have to do is add a source file called `Source.cpp`. Add the following code to it:

```
#pragma once

#include <string>
using namespace std;

class AudioDataHandler
{
public:
  AudioDataHandler();
  ~AudioDataHandler();
  string GetAudio(int data) // Set one of the enum values here from
the driver program
  {
    return Files[data];
  }

  enum AUDIO
  {
    NONE=0,
    BACKGROUND,
    BATTLE,
    UI
  };
private:
  string Files[] =
  {
```

```
        "",
        "Hello.wav",
        "Battlenn.wav",
        "Click.wav"
      }


    };

    int main()
    {
      AudioDataHandler _audioData;
      cout<<_audioData.GetAudio(AudioDataHandler::BACKGROUND);
    }
```

How it works...

In this example, we have created an audio data handler class. The class has `enum`, which stores all the logical names of the sounds, for example `battle_music`, `background_music`, and so on. We also have a string array, which stores the actual names of the sound files. The order is important and it has to match the order in which we have written `enum`. Now that this `enum` is created, we can create an object of this class and set and get the audio filename. The `enum` is stored as integers and starts at `0` by default, and the names serve as an index for the string array. So `Files[AudioDataHandler::Background]` is actually `Files[1]`, which is `Hello.wav`, and so the correct file will be played. This is a very neat way of organizing audio data files. The other way to handle audio in games is to have the names of the audio files and the attributes of their location in an XML or JSON file, and have a reader which parses this information and then fills up the array in the same way as we are doing. That way, the code is extremely data driven because the designer or the audio engineer can just change the values of the XML or the JSON file, without having to make any changes to the code.

13
Tips and Tricks

In this chapter, the following recipes will be covered:

- Effectively commenting your code
- Using bit fields in a struct
- Writing a sound technical design document
- Using the const keyword to optimize your code
- Using bit shift operators in an enum
- Using the new lambda feature in C++11

Introduction

C++ is a vast ocean. There are many concepts and techniques that are required to master C++. On top of that, there are also a few little tricks that a programmer can learn from time to time to help develop better software. In this chapter, we will look at some of the techniques that a programmer can learn to write better code.

Effectively commenting your code

Very often, a programmer is so engrossed in solving a problem that they forget to comment their code. Although this may not be a problem when they are working on it, if there are other team members involved who have to utilize that same section of code, it may become very difficult to fathom. Therefore, it is essential to comment code from an early stage of development.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will see how easy it is to comment code. Let's add a source file called `Source.cpp`. Add the following code to the file:

```
//Header files
#include <iostream>



class Game
{
  //Member variables (Already known)
public:
private:
protected:

};

//Adding 2 numbers
int Add(int a=4,int b=5)
{
  return a + b;
}



void Logic(int a,int b)
{
  if (a > 10 ? std::cout << a : std::cout << b);

}
int main()
{
  std::cout<<Add()<<std::endl;
  Logic(5,8);

  int a;
  std::cin >> a;
}
```

How it works...

Comments are supposed to be written on any section, to help fellow developers understand what is going on. To comment a code, we use the // double backslash symbols. Whatever we write within that will not be compiled and will be ignored by the compiler. As a result, we can use it to make notes on different aspects in the code. We can also use the /* and */ symbol to comment multiple lines. Anything that is within a pair of /* and */ symbols will be ignored by the compiler. This technique becomes useful if we need to debug an application. We first comment out a large section of the code that we think is the culprit. The code should now build. Then we start uncommenting the code till we reach a point where the code breaks again.

Sometimes programmers tend to over-comment. For example, there is no need to write `//Addition` on top of an addition function, as we can clearly see that two numbers are being added. Similarly, we should not under-comment. As there are no comments on top of the `Logic` function, we have no clue as to why we are using that function and what that function does. So we must remember to comment just enough. This will only happen with practice and by working in a team environment.

Using bit fields in a struct

In structures, we can use bit fields to denote what size we want the structure to be. As well as this, it is also important to understand what size a struct actually takes.

Getting ready

You need a Windows machine and a working copy of Visual Studio. No other prerequisites are required.

How to do it...

In this recipe, we will find out how easy it is to use bit fields to find the size of a struct. Add a source file called `Source.cpp`. Then add the following code to it:

```
#include <iostream>

struct Type
{
  int a;
  unsigned char c[9];
  unsigned  b;
```

```
  float d;

};

struct Type2
{
  int a : 2;
  int b : 2;
};
int main()
{
  std::cout << sizeof(Type)<<std::endl;
  std::cout << sizeof(Type2);

  int a;
  std::cin >> a;
}
```

How it works...

As you can see, in the example we have assigned a struct of int, a char array, an undefined unsigned variable, and a float. When we execute the program, the output should be the size of both the structures in bytes. Assuming we are running this program on a 64-bit machine, int is 4 bytes, unsigned char array is 9 bytes, unsigned by default is 4 bytes, and float is 4 bytes. If we add them up, the total is 21 bytes. But if we print it out, we will notice that the output is 24 bytes. The reason for this is called *padding*. C++ always fetches data in chunks of 4 bytes. Hence it will always pad with extra bytes till the size is a multiple of 4. Because the size of the struct came out at 21, the nearest multiple of 4 is 24, so we get that answer. Padding is not done to the structure as a whole, but per declaration, for example:

```
struct structA
{
   char a;
   char b;
   char c;
   int d;
};


struct structB
{
   char a;
   int d;
   char b;
```

```
    char c;
};

Sizeof structA = 2 bytes
Sizeof structb = 3 bytes
```

Looking at the second struct, what we have done is assigned a bit field. Although an int is 4 bytes, we can instruct it to just have 2 bytes. The syntax for doing it is adding a `:` symbol followed by the byte value. So for the second struct, if we find the value, it is going to output it as `4` instead of `8`.

Writing a sound technical design document

When we start a project, there are two documents that we generally rely on. The first document is a game design document, and the second is a technical design document. The technical design document should list the key features and high-level architecture of the key features. This system is changing rapidly though, with the advent of indie games. However, in a large-scale gaming studio, this process is still valid.

Getting ready

You need to have a working Windows machine.

How to do it...

In this recipe, we will see how easy it is to create a technical design document:

1. Open an editor of your choice, preferably Microsoft Word.
2. List the key technical components of the game.
3. Create a data flow diagram to represent the flow of data between various components of the engine.
4. Create a flowchart to explain the logic of a certain complex section.
5. Write pseudocode for the sections that are key to the development of the game.

How it works...

Once the key components are listed, the project manager can automatically assess the risk and complexity of each task. The developer will also understand what the key components of the engine or game are. This will help the developer plan their actions as well. When the data flow diagram is made, it will be easy to understand which component is dependent on which other component. As a result, the developer will know they have to implement *A* before they start coding *B*. A flow chart is also a great way to understand the flow of logic and sometimes helps to solve ambiguity that could occur in the future. Finally, pseudocode is essential for explaining to the developer how they must implement the code, or rather what is an advisable approach. As pseudocode is language independent, the same pseudocode could be used to write a game even in other languages apart from C++.

Using the const keyword to optimize your code

We have already seen in previous recipes that a `const` keyword is used to make data or a pointer constant so that we cannot change the value or address, respectively. There is one more advantage of using the `const` keyword. This is particularly useful in the object-oriented paradigm.

Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio.

How to do it...

In this recipe, we will find out how easy it is to use the `const` keyword effectively:

```
#include <iostream>

class A
{
public:

  void Calc()const
  {
    Add(a, b);
    //a = 9;       // Not Allowed
  }
  A()
  {
```

```
    a = 10;
    b = 10;

  }
private:

  int a, b;
  void Add(int a, int b)const
  {

    std::cout << a + b << std::endl;
  }
};

int main()
{

  A _a;
  _a.Calc();

  int a;
  std::cin >> a;

  return 0;
}
```

How it works...

In this example, we are writing a simple application to add two numbers. The first function is a public function. This mean that it is exposed to other classes. Whenever we write public functions, we must ensure that they are not harming any private data of that class. As an example, if the public function was to return the values of the member variables or change the values, then this public function is very risky. Therefore, we must ensure that the function cannot modify any member variables by adding the `const` keyword at the end of the function. This ensures that the function is not allowed to change any member variables. If we try to assign a different value to the member, we will get a compiler error:

```
error C3490: 'a' cannot be modified because it is being accessed
through a const object.
```

So this makes the code more secure. However, there is another problem. This public function internally calls another private function. What if this private function modifies the values of the member variables? Again, we will be at the same risk. As a result, C++ does not allow us to call that function unless it has the same signature of const at the end of the function. This is to ensure that the function cannot change the values of the member variables.

Using bit shift operators in an enum

As we have seen before in previous recipes, an enum is used to represent a collection of states. All the states are given an integer value by default, starting at `0`. However, we could specify a different integer value as well. More interestingly, we could use bit shift operators to club some of the states, easily set them to be active or inactive, and do other tricks with them.

Getting ready

To work through this recipe, you will need a machine running Windows with an installed Visual Studio.

How to do it...

In this recipe, we will see how easy it is to write bit shift operators in an enum:

```
#include <iostream>

enum Flags
{
  FLAG1 = (1 << 0),
  FLAG2 = (1 << 1),
  FLAG3 = (1 << 2)
};

int main()
{

  int flags = FLAG1 | FLAG2;

  if (flags&FLAG1)
  {
    //Do Something
  }
  if (flags&FLAG2)
  {
    //Do Something
  }


  return 0;
}
```

How it works...

In the above example, we have three flag states in the enum. They are represented by the bit shift operator. So in memory, the first state is represented as `0000`, the second as `0001`, and the third as `0010`. We can now combine the states by using the `OR` operator (`|`). We can have a state called `JUMP` and another state called `SHOOT`. If we want the character to now `JUMP` and `SHOOT` together, we can combine these states. We can use the `&` operator to check whether a state is active or not. Similarly, if we have to remove a state from a combination, we can use the `XOR` operator (`^`). We can disable a state by using the `~` operator.

Using the new lambda function of C++ 11

Lambda functions are the new addition to the C++ family. They can be described as anonymous functions.

Getting ready

To work through this recipe, you will need a machine running Windows and Visual Studio.

How to do it...

To understand a lambda function let's have a look at the following code:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
  vector<int> numbers{ 4,8,9,9,77,8,11,2,7 };
  int b = 10;
  for_each(numbers.begin(), numbers.end(), [=](int y) mutable->void {
if(y>b) cout<<  y<<endl;  });

  int a;
  cin >> a;

}
```

How it works...

Lambda functions are a new addition to the C++11 family. They are anonymous functions and can be very handy. They are generally passed as arguments to a function. The syntax of a lambda function is as follows:

- `[ capture-list ] ( params ) mutable(optional) exception attribute -> ret { body }`

The `mutable` keyword is optional and is used to modify the parameters and call their non-const functions. The attribute provides the specification of the closure type. The capture list is optional and has a list of allowed types:

- `[a,&b]`: Here `a` is captured by value and `b` is captured by reference
- `[this]`: This captures the `this` pointer by value
- `[&]`: This captures all automatic variables used in the body of the lambda by reference
- `[=]`: This captures all automatic variables used in the body of the lambda by value
- `[]`: This captures nothing

Params are lists of parameters, as in named functions, except that default arguments are not allowed (up to C++14). If auto is used as a type of a parameter, the lambda is a generic lambda (since C++14). `ret` is the return type of the function. If no type is provided, then `ret` tries to auto-inject a return type or void if it is not returning anything. Finally, we have the body of the function, which is used to write the logic of the function.

In this example, we store a vector list of numbers. After that, we traverse the list and use a lambda function. The lambda function stores all the numbers that are greater than 10 and displays the number. Lambda functions can be difficult to start off with but, with practice, they are very easy to grasp.

Index

Symbols

2D game
creating 246-250
3D game
creating 250-252

A

abstract factory method
using 165-169
algorithm
about 89
complexity, finding 95-97
Apache Subversion (SVN) 6
application layer, OSI model 278
area of interest filtering
using 301
arguments
passing, to thread 266, 267
Artificial intelligence (AI)
about 201
adding, to game 202, 203
artificial neural networks (ANN)
using 222-227

B

background music
adding 306-308
behavioral movements
adding 219-222
Berkley Sockets (BSD) 287
Binary Space Partition (BSP) Tree
using 205-208
bit fields
using, in struct 317-319
bit shift operators
using, in enum 322
bitwise operations
used, for advanced checks 21
used, for optimization 21, 22
Box2D
about 233
installing 245, 246
integrating 245, 246
URL 246
branch
creating 198
broad-phase collision 245
BSP 245
bubble sort 92, 93
Bullet Physics SDK 233

C

C++
lambda function, using 323, 324
call stacks
using, for memory storage 7, 8
classes
used, for data abstraction 26-30
used, for data encapsulation 26-30
code
commenting 315-317
optimizing, const keyword used 320, 321
reusing, polymorphism used 30-33
collide 239, 244
command design pattern
using 184-187
compiler 4
complex program
breaking down, dynamic programming used 99-101

Concurrent Versions System (CVS) 6
conflicts
resolving 197
const keyword
using, to optimize code 320, 321
controls
using 131-135
control tool
right source control tool, selecting 5, 6
copy constructors
using 34-36

D

data
receiving 294-297
saving, by selecting host 193-195
sending 289-294
storing, graphs used 81, 82
storing, linked lists used 70-72
storing, queues used 75, 76
storing, stacks used 72-75
storing, STL hash tables used 86-88
storing, STL lists used 83-85
storing, STL maps used 85, 86
storing, trees used 77-80
data abstraction
classes, using 26-30
data encapsulation
classes, using 26-30
datagrams 277
data link layer, OSI model 277
data race 269-271
data structures
about 59
advanced data structures, using 60-70
datatypes
casting between 14-16
deadlocks
avoiding 268, 269
debugger 5
decision making AI
creating 208, 219
decision tree 208, 219
design patterns
abstract factory method, using 165-169
command design pattern, using 184-187
flyweight pattern, using 174-178
factory method, using 162-165
observer pattern, using 170
singleton design pattern, using 160-162
strategy pattern, using 179-184
using 187-190
device
endian-ness, finding 97-99
dialogs
using 131-135
Diff tool 197
Dijkstra's 204
divide and conquer algorithms
used, for solving problems 102-105
dynamic allocation
used, for managing memory 16-19
dynamic programming
used, for breaking down complex program 99-101

E

endian-ness
finding, for device 97-99
enum
bit shift operators used 322, 323
evolutionary algorithm (EA) 227

F

factory method
using 162-165
files
used, for input and output 48-51
flyweight pattern
using 174-178
FMOD
installing 306
URL 306
function
reusing, function overloading used 45-48

function overloading
used, for reusing functions 45-48

G

game
advanced game, design patterns used 187-190
Artificial intelligence (AI), adding 202, 203
concurrency 264, 265
creating 52-54
heuristics, using 203-205
physics rule, using 234-238
ragdoll, using 254, 261
socket programming, using 286-288
Genetic algorithms (GA)
cross-over 229
fitness function 229
initial population 229
mutate 230
using 227-229
GIT 6
Graphics device interface (GDI)
Windows resources, using with 126-130
graphs
used, for storing data 81-83
greedy algorithms
used, for solving problems 101, 102
GUI interface builder 5

H

handles
using 109-114
heap sorting algorithm 105
heuristics
using, in game 203-205
host
selecting, to save data 193-195

I

implicit conversions 14
input layer 227
insertion sort 92, 93
integrated development environment (IDE)
installing, on Windows 2-5
items
arranging, sorting techniques used 90-93
looking for, searching techniques used 93-95

K

keyboard
adding, with text output 118, 126

L

lag
dealing with 297-299
lambda function
of C++, using 323, 324
Last In First Out (LIFO) data structure 8, 58, 72
linked lists
used, for storing data 70-72
linker 5
LLC layer (Logical Link Control) 277
local perception filter
URL 304
using 302, 303

M

MAC layer (Media Access Control) 277
memory
error messages 20
managing, dynamic allocation used 16-20
memory addresses
storing, pointers used 11-13
memory handling 2
memory storage
call stacks used 7, 8
Mercurial 6
mouse controls
adding, with text output 118, 126
multiple sound file names
dealing with 312, 313
mutex 269-271

N

narrow-phase collision 245
network layer, OSI model 277
NLP (Natural Language Processing) 203

O

observer pattern
using 170-173
OOP (Object-oriented programming) 25, 26
OpenGL 233
operator overloading
used, for reusing operators 36-45
operators
reusing, operator overloading used 36-45
OSI (Open Systems Interconnection) model, layers
about 276, 277
application layer 278
data link layer 277
network layer 277
physical layer 277
presentation layer 278
session layer 278
transport layer 278

P

packets
serializing 281-285
particle system
creating 252-254
physical layer, OSI model 277
physics rule
using, in game 234-238
pointers
URL 14
used, for storing memory addresses 11-13
polymorphism
used, for reusing code 30-33
presentation layer, OSI model 278
problems
solving, divide and conquer algorithms used 102-105
solving, greedy algorithms used 101, 102
protocol
selecting 278-280
PvP (player versus player) 276

Q

queues
used, for storing data 75, 76
quick sort 92, 93

R

ragdoll
using, in game 254, 261
Ragdoll physics 254
recursions
using, cautiously 9, 11
references
URL 14
Resource Acquisition is Initialization technique (RAII) 271
revision control software 1

S

searching techniques
used, to look for item 93-95
selection sort 92, 93
session layer, OSI model 278
singleton design pattern
using 160-162
sorting techniques
used, for arranging items 90-93
sound effects
adding 308, 309
manager, creating 309-311
source control
adding 195, 197
code, committing 195-197
code, updating 195, 196
types, URL 192
URL, for GIT client 192
URL, for Mercurial client 192
URL, for SVN client 192
versions 192
sprites
animated sprites, using 155-158
using 136-155

stack frame 8
stacks
about 8
used, for storing data 72, 75
stack unwinding 271
state-machine design pattern 222
STL hash tables
used, for storing data 86-88
STL lists
used, for storing data 83-85
STL maps
used, for storing data 85, 86
strategy pattern
using 179-184
struct
bit fields, using 317-319
synchronized simulation
using 299, 300

T

technical design document
writing 319, 320
templates 55-57
text output
keyboard, adding 118-126
mouse controls, adding 118-126
thread
arguments, passing 266, 267
creating 264, 265
detaching 265, 266
joining 265, 266
thread-safe class
writing 271-273
Tortoise SVN
about 192
URL 193
transfer control protocol (TCP) 278
transport layer, OSI model 278
trees
used, for storing data 77-80

U

user datagram protocol (UDP) 278

V

versioning client
installing 192, 193
virtual method table (VMT or Vtable) 33
Visual Studio 1
Visual SVN Server
URL 194
VLD
URL 19

W

waypoint systems
using 231, 232
window
first window, creating 114-118
IDE, installing 2-5
Windows classes
using 109-113
Windows game
creating 108, 109
message box, types 109
Windows resources
using, with Graphics device interface (GDI) 126-130
Win Merge 197
Wwise 306
WYSIWYG tool editor 5

X

XML serializer
URL 285