1

Game Development Basics

In this chapter, we will cover the basic concepts that you need to know to kick-start your career in games development. You will learn how to:

* Install IDE on Windows
* Choose the right source-control tool
* Use call-stacks for memory storage
* Use recursions cautiously
* Use pointers to store memory address
* Cast between various data-types
* Manage memory more effectively using dynamic allocation
* Use bitwise operations for advanced check and optimisation

# Introduction

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

A 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 which have worked upon 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 of how memory must be assigned depending on the current usage and need of the program. As a result of that, we can allocate memory when there is a need and de-allocate them accordingly.

If we do not de-allocate memory, we might run out of memory very soon especially if we are using recursion. We do however have pointers at our disposal for dynamically allocation memory.

Sometimes there is a need to convert from one data type to another to prevent loss of data, to pass the correct data type in a function, etc. C++ provides us few ways by which we can do those castings.

The recipes in this chapter will primarily focus on the above mentioned topics and deal practical ways to implement them.

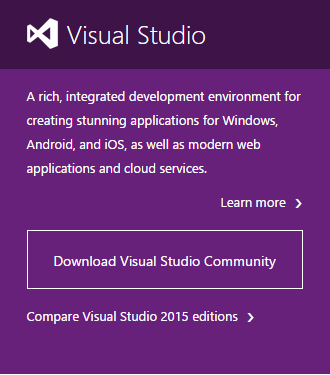
# Install IDE on Windows

Throughout the following recipes, we will use Visual Studio as our main IDE to build any kind of application or to explain any kind of technique. So you should first install the IDE on your windows computer to get things started.

## Getting ready

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

## How to do it...

1. Go to <https://www.visualstudio.com>
2. Click on the Download Visual Studio Community
3. 
4. Illustration 1: Download Visual Studio Community
5. This should download an .exe
6. After it downloads, double click to start the installations
7. Make sure you have all the updates necessary on your Windows machine.
8. You can also download any version of Visual Studio or Visual C++ express.
9. If the application asks for a starting environment settings, select C++ from the available options
10. Note: There are other free IDEs for C++ like NetBeans, Eclipse, and Code::Blocks.
11. Note: While Visual Studio works only for Windows, Code::Blocks and other IDEs such are cross platform, can work on Mac and Linux as well
12. For the remaining on this chapter, all code examples and snippets will be provided using Visual Studio.

## How it works...

An IDE is a programming environment. An IDE can consists of various functionalities and 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 interface 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, etc. An example snapshot of the Visual Studio code editor is given below

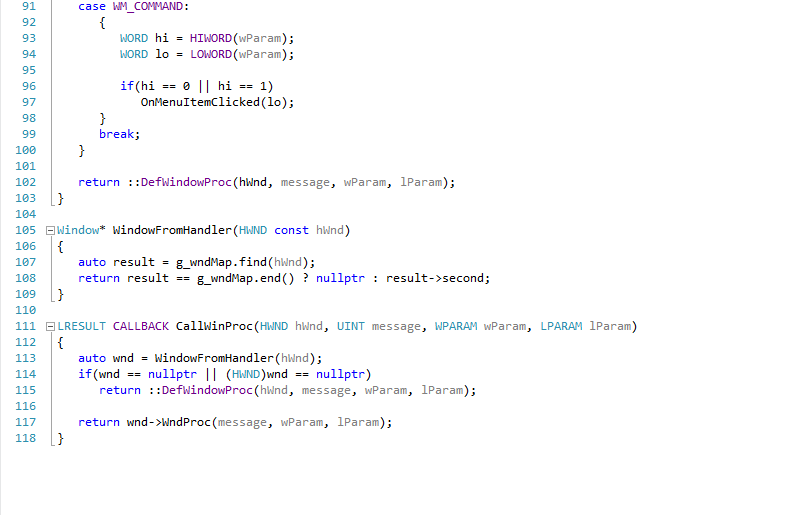


Illustration 2: Visual Studio Code Editor

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.

Illustration 2: Visual Studio Code Editor

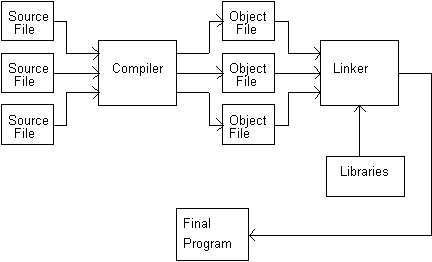


Illustration 3: Compiler and Linker

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

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

# Choose the right source-control tool

I cannot stress enough the importance of backing up your source code. I have heard so many time programmers telling me that they have lost 2-3 months of work because their hard disk crashed and they had no backup. Advantage of having a backup to a central server is that you will never loose work, you can download the code on any other machine and you also go back to any of your changes from the past. Imagine it like a checkpoint which we have in games and you go back to that checkpoint if you face problems.

## Getting ready

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

## How to do it...

1. First analyse what are the choices that are available to you.
2. The choices primarily include CVS, SVN, Mercurial and GIT.

## How it works...

## CVS has been around for a long time. So there are tons of documentation and help. However a lack of atomic operations leads to source corruption easily 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 drawback of this tool is that is comparatively very slow in its operations.

## GIT was made primarily for Linux but it enhances the operation speed a lot. It does work on UNIX systems as well. It has cheap branch operations but it is not totally optimised for single developer and its Windows support is limited as compared to Linux. However GIT is extremely popular and many prefer GIT over SVN or CVS.

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

## So to sum up in the end, use SVN if you want a central repository which others can push and pull. Although it has its limitations, it’s easy to learn. Use Mercurial and GIT if you want a distributed model. In this case, there is a repository in 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 easy to learn than GIT.

# Use 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 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.

## Getting ready

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

## How to do it...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a 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 CounTotalBullets, 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 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 to hold data for each function in your program. 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 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 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.

When your program calls a function, 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.

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

## See also

* Manage memory more effectively using dynamic allocation

# Use 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 function call. You need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a 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 RecursiveFactorial(int number)

{

int iCounter = 1;

if (number < 2)

{

return 1;

}

else

{

while (number>0)

{

iCounter = iCounter\*number;

number -= 1;

}

}

return iCounter;

}

int Factorial(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 above piece of code, both the functions find the factorial of a number. However when it is found 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. In 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 create a deadlock or crash.

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

## There's more...

* You can use a larger data-type than int if you are finding the factorial of a number greater than 15 as the resulting factorial will be quite large to be stored in int.

# Use pointers to store memory address

In the previous two chapters we have seen how not having sufficient memory can be a problem to us. However till now we have had no control as to how much memory is assigned and what is assigned to each memory address. Using pointers we can address this issue. In my opinion, pointers is the single most important topic of 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 start 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...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a 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 variable: "<<pfLocalCurrentHealth << endl;

cout << "Address of the Local variable: "<<&pfLocalCurrentHealth << endl;

cout << "Value at the address of the Local variable: "<<\*pfLocalCurrentHealth << endl;

\_getch();

return 0;

}

## How it works...

One of the most powerful tools of a C++ programmers is to manipulate computer memory directly. A pointer is a variable that holds a memory address. Each variable and object used in C++ program is stored in a specific place in memory. Each memory location has a unique address. Memory addresses will vary depending on Operating System. Amount of bytes taken up depends on variable type.i.e. float = 4 bytes, short = 2 bytes.

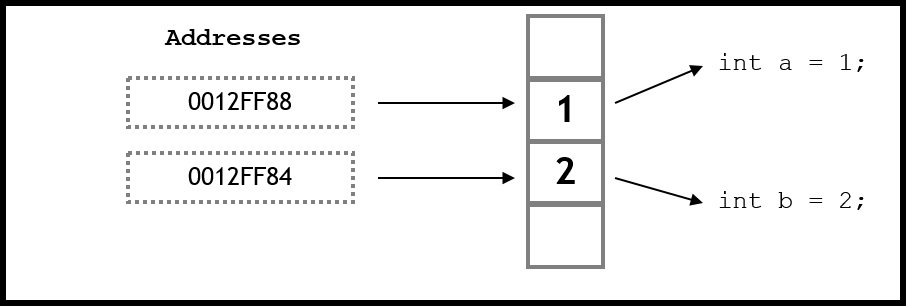


Illustration 4: Pointers and memory storage

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

## There's more...

### How to use const pointer

Let us consider the following declarations; const float\* pfNumber1, float\* const pfNumber2, and const float\* const pfNumber3. All of the above 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.

# Cast between different data types

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 data types. 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 explicitly 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 affect our code adversely. 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. C++ provides us with more fine controls.

## Getting ready

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

## How to do it...

## Open Visual Studio.

## Create a new C++ project

1. 3. Select a 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 inton int and saves in

//into a float

fNumber = iNumber/2;

cout << "Number is " << fNumber<<endl;

//C-style casting.Not reccomended 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, dynamic\_cast. Now we are going to look into static\_cast. We will look into the remaining 3 casting technique after we discuss dynamic memory and classes. Converting from a smaller data type to a larger type is called promotion and is guaranteed to have no data loss. However conversion from a higher data type 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 example above. We have initialised an integer with the value 5.

Next we have initialised a floating point variable and storing 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 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 static\_cast operation is static\_cast<resultant\_data\_type> (expression). The compiler checks if that casting conversion is safe and then executes the type casting operation.

## See also

* Advanced type casting

# Manage memory more effectively using dynamic allocation

Programmers generally deal with five areas of memory: Global name space, Registers, Code space, Stack and the Free Store. When an array is initialised, the number of elements have to be fined. This leads to lots of memory problems. Most of the time, all elements are not used that we allocated and sometimes we need more elements. To help overcome this problem, C++ facilitates memory allocation while an EXE is running, by using the free store.

The Free Store is a large area of memory that can be used to store data. Sometimes referred to as “the heap”. We can request some space on the Free Store, and it will give us an address in which 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 programmers’ responsibility to free any Free Store memory used by their program.

The advantages of the free store is that there is no need to pre-allocate all variables. We can decide at run-time 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

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

## How to do it...

## Open Visual Studio.

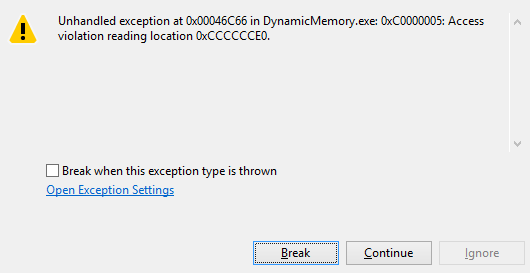
## Create a new C++ project

## Add a source file called main.cpp or anything that you want to name the source file.

## Add the following lines of code.

1. #include <iostream>
2. #include <conio.h>
3. #include <string>
4. using namespace std;
5. int main()
6. {
8. int iNumberofGuns, iCounter;
9. string \* sNameOfGuns;
10. cout << "How many guns would you like to purchase? ";
11. cin >> iNumberofGuns;
12. sNameOfGuns = new string[iNumberofGuns];
13. if (sNameOfGuns == nullptr)
14. cout << "Error: memory could not be allocated";
15. else
16. {
17. for (iCounter = 0; iCounter<iNumberofGuns; iCounter++)
18. {
19. cout << "Enter name of the gun: ";
20. cin >> sNameOfGuns[iCounter];
21. }
22. cout << "You have purchased: ";
23. for (iCounter = 0; iCounter<iNumberofGuns; iCounter++)
24. cout << sNameOfGuns[iCounter] << ", ";
25. delete[] sNameOfGuns;
26. }
27. \_getch();
28. return 0;
29. }

## How it works...

* 1. You can allocate memory on the free Store using the new keyword. newis 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. Hence 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 if the pointer allocation has been successful, otherwise we will trying to access a part of memory which has not been allocation and we may get an error from the compiler as shown in the figure below and your application will crash.
  2. 

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.When the function that the pointer is declared in goes out of scope, the memory on the free store is not automatically deallocated. 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 operator delete[] 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 which have not been deallocated and remember memory is a finite space, so eventually you are going to run into trouble.

When we use the 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 if you only know the pointer sNameOfGuns. There are two popular techniques that do this. Both these are used by commercial compilers, both   
have tradeoffs, neither are perfect.

* Technique 1:

Over allocate the array and put the number of items just to the left of the first element. Faster of the two techniques. More sensitive to the problem of the programmer saying delete sNameOfGuns, instead of delete[] sNameOfGuns.

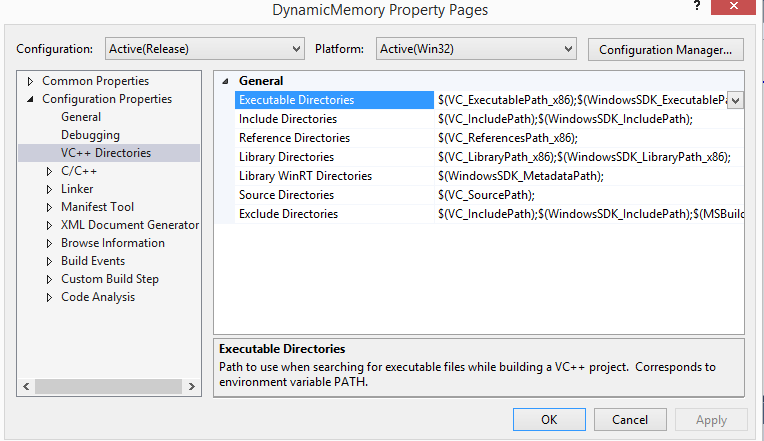
* Technique 2:

Use an associative array with the pointer as a key and the number of items as the value. Slower of the two techniques. Less sensitive to the problem of the programmer   
saying delete sNameOfGuns, instead of delete[] sNameOfGuns.

## There's more...

### Detecting memory leaks with VLD

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 include directory for vld as shown in the figure below.



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 if there are any memory leaks in your application.

### Understanding the error messages

* 1. When using the debug build, one may notice the following values in memory during debugging:
* 0xCCCCCCCC

Allocated on the stack, but not yet initialized.

* 0xCDCDCDCD
  + - 1. This means memory has been allocated in the heap, but it is not yet initialized. (Clean Memory)
* 0xDDDDDDDD
  + - 1. This means memory has been released from the heap. (Dead Memory)
* 0xFEEEFEEE
  + - 1. Deallocated from the Free Store.
* 0xFDFDFDFD
  + - 1. “No Mans 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 programming is trying to access memory at an index outside of an array’s max size.

# Using bitwise operations for advanced checks and optimisation

In most cases, a programmer will not need to worry too much about bits unless there is a need for writing 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

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

## How to do it...

## Open Visual Studio.

## Create a new C++ project

## Add a source file called main.cpp or anything that you want to name the source file.

## Add the following lines of code.

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. void Multi\_By\_Power\_2(int iNumber, int iPower);
5. void BitwiseAnd(int iNumber, int iNumber2);
6. void BitwiseOr(int iNumber, int iNumber2);
7. void Complement(int iNumber4);
8. void BitwiseXOR(int iNumber,int iNumber2);
9. int main()
10. {
11. int iNumber = 4, iNumber2 = 3;
12. int iPower = 2;
13. unsigned int iNumber4 = 8;
14. Multi\_By\_Power\_2(iNumber, iPower);
15. BitwiseAnd(iNumber,iNumber2);
16. BitwiseOr(iNumber, iNumber2);
17. BitwiseXOR(iNumber,iNumber2);
18. Complement(iNumber4);
19. \_getch();
20. return 0;
21. }
22. void Multi\_By\_Power\_2(int iNumber, int iPower)
23. {
24. cout << "Result is :" << (iNumber << iPower)<<endl;
25. }
26. void BitwiseAnd(int iNumber, int iNumber2)
27. {
28. cout << "Result is :" << (iNumber & iNumber2) << endl;
29. }
30. void BitwiseOr(int iNumber, int iNumber2)
31. {
32. cout << "Result is :" << (iNumber | iNumber2) << endl;
33. }
34. void Complement(int iNumber4)
35. {
36. cout << "Result is :" << ~iNumber4 << endl;
37. }
38. void BitwiseXOR(int iNumber,int iNumber2)
39. {
40. cout << "Result is :" << (iNumber^iNumber2) << endl;
41. }

## How it works...

## The leftshift operator is the equivalent of moving all the bits of a number a specified number of places to the left. In our example, the number 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, 3 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, i.e. 4\*2^3 which is again 16. Similarly the right shift operation is equivalent to integer division by 2^shift\_arg.