13

Tips and Tricks

In chapter, the following recipes will be covered:

* Effectively commenting your code
* Using bit fields in struct
* Using const keyword to optimise your code
* Writing a sound technical design document
* Using bitshift operators in 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 few little tricks which a programmer can learn from time to time to help in developing better software. In this chapter we will look at some of the techniques which a programmer can learn to help write better code.

# Effectively commenting your code

1. Very often a programmer is so engrossed in solving a problem, that he forgets to comment his code. Although this may not be a problem when he is working on it, however if there are other team members involved who have to utilise that same section of code it may get very difficult to fathom. Hence it is extremely essential to comment code from an early stage of development.

## Getting ready

To step 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.

* Add a source file called Source.cpp

//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 make fellow developers more familiar with what is going on. To comment in a code, we use the “//” double back-slash symbols. Whatever we write within that will not be compiled and will be ignored by the compiler. As a result, we can use that to make a note on different aspects in the code. We can also use the /\* symbol to comment about multiple line. Anything that is within a /\* and \*/, 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 which 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 also 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 practise and working in a team environment.

# Using bit fields in struct

In structures, we can use bit fields to denote how much size we want the structure to be. Along with this, it is also important to understand how much size a struct actually take.

## Getting ready

You need a Windows machine and a working copy of Visual Studio. No other pre-requisite is needed.

## How to do it...

In this recipe we will find out how easy it is to use bit fields and find the size of a struct.

* Add a source file called Source.cpp

Code Snippet

#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 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 structure 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. Still 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 up with extra bytes till the size is a multiple of 4. Because the size of the struct came out to be 21, the nearest multiple of 4 is 24. Hence we get that answer.

Looking at the second struct, what we have done is assigned 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 “:” and then 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 2 backbones that we generally rely on. This system is changing rapidly though with the advent of indie games. However, in a large scale gaming studio, this process is still valid. The first document is a game design document. The second document is a technical design document. The technical design document should list the key features and high level architecture of the key features.

## Getting ready

1. 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

* Open an editor of your choice, preferably Microsoft Word.
* List the key technical components of the game.
* Create a data flow diagram to represent the flow of data between various components of the engine.
* Create a flowchart to explain the logic of a certain complex section.
* Write pseudo code for the sections that are key to the development of the game.

## How it works...

Once the key components are listed, automatically the project manager can assess the risk and complexity of each task. Accordingly, the developer will also be at ease at understanding what the key components of the engine/game is. This will help the developer plan his 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 he has to implement A before he starts coding B. A flowchart is also a great way to understand the flow of logic and sometimes helps to solve ambiguity which could have happened in the future. Finally, a pseudo code is essential to explain to the developer how he must implement the code or rather what is an advisable approach. As a pseudo code is language independent, the same pseudo code could be used to write a game even in other languages apart from C++.

# Using the const keyword to optimise code

We have already seen in previous recipes that a const keyword is used to make a data or 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 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...

1. In this recipe we will find out how easy it is to use the const keyword effectively.
2. #include <iostream>
3. class A
4. {
5. public:
6. void Calc()const
7. {
8. Add(a, b);
9. //a = 9; // Not Allowed
10. }
11. A()
12. {
13. a = 10;
14. b = 10;
15. }
16. private:
18. int a, b;
19. void Add(int a, int b)const
20. {
22. std::cout << a + b << std::endl;
23. }
24. };
25. int main()
26. {
28. A \_a;
29. \_a.Calc();
30. int a;
31. std::cin >> a;
32. return 0;
33. }

## 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 it is 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 would have been a very risky function. Hence we must ensure that the function cannot modify any member variables. Hence we add 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 assigning 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 that 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 even that function cannot change the values of the member variables.

# Using bit shift operators in 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 non-active and do other tricks with them.

## Getting ready

To step 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 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 3 flag states in the enum. They are represented by the bit shift operator. So in memory, the first state is represented as 0000, second as 0001, thirds as 0010. We can now combine the states together 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 states like that. 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 lamba function of C++

Lamda functions are the new addition of the C++ family. They can be described as anonymous functions.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

#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 to modify the parameters and to 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.

1. [a,&b] where a is captured by value and b is captured by reference.
2. [this] captures the “this” pointer by value
3. [&] captures all automatic variables used in the body of the lambda by reference
4. [=] captures all automatic variables used in the body of the lambda by value
5. [] captures nothing

Params are the list of parameters, as in named functions, except that default arguments are not allowed (until 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 it 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 are storing a vector list of numbers. After that we are traversing the list and using 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 can with practise, they are very easy to grasp and understand.