**Overview of Basic Concepts in C++ - Chapter 5**

Table of Contents

[Reference Arguments 3](#_Toc49598493)

[Reference Returns 4](#_Toc49598494)

[Function Overloading 5](#_Toc49598495)

[Inline Functions 6](#_Toc49598496)

[Default Arguments 7](#_Toc49598497)

[const Function Arguments 8](#_Toc49598498)

[Scope and Storage Class 9](#_Toc49598499)

Chapters 2 – 5 discuss some basic concepts about C++. All of the details in the book are not being covered here, only those that are new or important.

## Reference Arguments

When we pass an argument to a function, the function creates a new variable and copies the value of the argument onto that variable. In this way, it is unable to access the original variable in the program. However, if we pass a reference to that variable to that function, it allows the function to access the original variable and edit it.

#include<iostream>  
using namespace std;  
  
void edit (int& a)  
{  
 a++;  
}  
  
int main()  
{  
 int a = 2;  
 edit(a);  
 cout << a;  
}

C++

To pass a variable as a refence argument, we add a & sign after the variable type in the declaration or definition, as show above. The above code edits the value of an integer that is outside the scope of the function editing it. This could also be done with a structure since the structure name acts as a custom variable type.

Reference arguments are particularly helpful when two values must be edited by the same function, since a function can only return one value. Note however, that this is different from a pointer, which will be discussed in Chapter 10. A reference is the same variable, just with a different name. A pointer is the address to a variable.

## Reference Returns

It is also possible to return values by reference. This can allow us to avoid copying some large data, and it also allows us to use a function call on the left-hand side. This is a very weird concept, so an example should help.

#include<iostream>  
using namespace std;  
  
int x;  
  
int& setx()  
{  
 return x;  
}  
  
int main()  
{  
 setx() = 92;  
}

C++

The function setx() has a return type int&, so the integer it returns is actually a reference to a variable. Until this, the concept is pretty simple.

The next thing we do is in the main function, with the line setx() = 92. The variable that the function setx() returns, x, is given a value of 92. Whereas normally the result of a function is a value, here, it is a variable. This also means that a function that returns a reference cannot return a constant value, only a variable. It also has to be a reference to a global variable.

The usage of this technique will be made clear while studying Operator Overloading in Chapter 8.

## Function Overloading

There are some functions that can work with different types of arguments, such as the cout function which prints out whatever is passed to it. They make use of a property known as overloading. In essence, the same function is defined multiple times with different numbers and/or types of arguments and the compiler uses the function based on which arguments are passed to it.

#include<iostream>  
using namespace std;  
  
void print (int a)  
{  
 cout << a;  
}  
void print (char b)  
{  
 cout << b;  
}  
  
int main()  
{  
 print (2);  
 print ('a');  
}

C++

## Inline Functions

When a function is called, the program jumps to the function, executes the code, and then comes back. This save memory since we can use the same bit of code repeatedly. However, jumping around and storing variables takes time. In some cases, it is not worth the memory saved to call the function in this way. It would actually be better to simply insert the code of the function into the place that it is required. However, this would make the code more complicated. To solve this, we can use inline functions.

An inline function is used in exactly the same way as a normal function. The only difference is, during the declaring of the function, the keyword inline is used before the return type of the function. For example,

inline int add();

C++

It should be noted that one- or two-line functions are the best candidates to be used as inline functions. Even then, the compiler may choose to ignore the request to make a function inline.

Inline functions serve the same purpose that macros did in C, i.e. the program seems certain words and replaces it with some code. Inline functions are better than macros however, since they are simpler to use.

## Default Arguments

Functions can be declared with default values. For example,

void print (int a = 2, int b = 3);

C++

This will allow us to call the function with fewer arguments, since the default values will be used in place of the missing arguments. However, we cannot leave out arguments from in between, only the end. For example, in a function with 5 arguments, we could leave out the last 4, only declaring the first, but we cannot leave out just the third argument, since the compiler has no way of knowing then which argument was left out.

Default arguments can be used in scenarios where the arguments almost never change. They can also be used when the number of arguments in a function is changed after a larger part of the program has been written, which means the function calls that have already been made do not need to be edited.

## const Function Arguments

If an argument is being passed to a function by reference, it may sometimes be important to make sure that the function is unable to modify it. This can be done by using the const keyword while passing the argument like:

void aFunc (int& a, const int& b);

C++

Here, the integer a can be modified by the function, but the integer b cannot.

If a const variable is being passed by reference to a function, then it *has* to be declared const in the argument. This is not required if the argument is being sent as a value however, since the function will be unable to modify the value anyway.

## Scope and Storage Class

The scope of a variable determines which parts of a program can access it. There are two types of scopes (for now), local and file. Variables with a local scope are only visible within the block in which they are defined. Any variables declared inside a function have a local scope. Variables declared outside of any function are known as global variables.

The storage class of a variable determines how long it can stay in existence. There are two types of storage class, automatic and static. Automatic variables only exist while the function they are a part of is running. Static variables exist as long as the program is running. By default, any variable declared within a function is automatic and any global variables are static.

A local variable can be made static by using the keyword static while declaring the variable. An important note here is, the local variable does not exist until the function it exists in is called for the first time. After that function has finished executing, it continues to exist for the duration of the program, whereas normally it would have been destroyed.

The initialization of a static local variable only takes place the first time the function is run, unlike automatic local variables which are initialized every time the function is run.