An Introductory Guide

For Beginners

Francis John Thottungal

C++

An Introductory Guide

For Beginners

Francis John Thottungal

C++

Booktango

1663 Liberty Drive

Bloomington, IN 47403

www.booktango.com

877-445-8822

ISBN: 978-1-4689-6322-9 (ebook)

Contents

Overview

C++ is generally a language taught to technical students in college around the world. The main reason C++, a derivative of C, has become important is that it is used in pretty much anything of a serious kind in programming such as developing drivers, graphical user interfaces, operating systems etc.

It is a technical language apt for technologists. However, anyone who is interested in programming can master it with some effort. This language has two parts to it- standalone and object oriented.

The approach:

In this guide, the approach is to emphasize the basic building blocks of a C++ program. The emphasis is to help one to make simple standalone programs. The examples in this guide are relatively small and perhaps academic.

One can download the zip file of all example code used in this guide at the location given at the end of the book.

Software Used:

There are many C++ editors in the market. I used the Microsoft Visual Express 2010.

To use this software or its latest version, just install the express version or full version after download from http://www.microsoft.com.

You may want to do a search for editors especially if use non-windows based computers such as Linux or Mac.

As this is not a textbook, the goal of this guide is just for a quick start to understanding how to write basic C++ code. It is intended for those completely new to the subject or programming in general. It might also serve those who prefer less technical jargon and simple examples to get a grasp of the language.

You may view or download the instructions and screen shots for using Microsoft Visual Studio or Express versions at the website stated at the end of the book.

1

The first program

"Come forth into the light of things, let nature be your teacher."
— William Wordsworth

1.0 The first program

The first program usually written in any programming language class is one that displays the word "Hello World". This is a kind of an unwritten tradition. Let us write the code for that now:

```
#include <iostream>
int main()
{
std::cout << "Hello World!";
}
```

In the above code:

A hash sign (#) indicates directives read and interpreted by the *pre-processor*.

The int main() is the declaration of a function. All C++ programs have a main function.

The open brace ({) indicates the beginning of main's function definition, and the closing brace (}), indicates its end.

The std::cout << "Hello World!" is a C++ statement to be executed.

Here << means insertion and cout is the keyword that is needed to output the text

All statement ends with a semicolon (;). One can enter blank lines between lines to make the code readable.

In C++, one can write several statements in a single line, or each statement can be in its own line.

Now, let us add an additional statement to our first program. The output of the code is *Hello World! My C++ program.* In other words, the two statements are on the same line.

To avoid this use endl at the end of each statement.

```
#include <iostream>
int main()
{
 std::cout << "Hello World!";
```

```
 std:: cout << “My C++ program”;
}
```

```
#include <iostream>
int main()
{
 std::cout << “Hello World!”; << endl;
 std:: cout << “My C++ program”; << endl;
}
```

1.1 Comments

C++ supports two ways of commenting code:

```
// line comment
/* block comment */
```

One can write short comments using the line comment. The block comments are used when larger amount of text is required in the comments.

Let us add comments to our “Hello World” program:

We will use block comments at the beginning of the program and line comments for the statements.

```
/*This is a block comment. This program is written to show the usage of comments- block and line*/
#include <iostream>
int main ()
{
 std::cout << “Hello World!”; // prints Hello World!
}
```

1.2 Using namespace: std

By using the namespace std we can avoid writing std::cout everytime we want something written to the screen. For example:

```
/* A program to write Hello World. This demonstrates the use of block comments*/
#include <iostream>
using namespace std;
int main ()
```

```
{
 cout << “Hello World!”; // prints Hello World!
}
```

1.3 Variables

Variables can be found in mathematics. You must have seen statements such as:

A= 1; or a =5; or a+b= c etc.

Variables hold values or data for a period and reference a particular area of memory. Like many other programming languages, C++ uses variables. Each variable should have a distinct name or identifier. Instead of calling a variable by the letters of the alphabet, we could give the variable useful names such as result as long as the name is not reserved as an exclusive C++ identifier.

A *valid identifier or variable name* is a sequence of one or more letters, digits, or underscores characters (_). Spaces, punctuation marks, and symbols cannot be part of an identifier. In addition, identifiers shall always begin with a letter. They can also begin with an underline character. In no case can it begin with a digit.

C++ language is case sensitive so a capital variable identifier is different from lowercase variable identifier.

Some examples of reserved identifiers in C++ are:

The list is only a partial one.

asm	else	new
auto	enum	operator
bool	explicit	private
break	export	protected
case	extern	public
catch	FALSE	register
char	float	reinterpret_cast
class	for	return
const	friend	short
const_cast	goto	signed
continue	if	sizeof

default	inline	static
delete	int	static_cast
do	long	struct
double	mutable	switch

Specific compilers may have additional reserved words.

1.4 Fundamental Data Types

- **Character types:** They can represent a single character, such as ‘A’ or ‘$’. The most basic type is char, which is a one-byte character. (ex:Char)
- **Numerical integer types:** They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can be either *signed* or *unsigned*, depending on whether they support negative values or not. (ex: int)
- **Floating-point types:** They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used. (ex: float, double)
- **Boolean type:** The Boolean type, known in C++ as bool, can only represent one of two states, true or false. (ex: bool)

1.5 Declaration of Variables

To declare a variable of type integer we write as follows:

```
int a;
int b;
float firstnumber:
double w;
char a;
```

We can also declare more than one variable of the same type in a single line as follows:

```
int x, y, z;
```

One can initialize variables.

```
int x=0;
```

```
#include <iostream>
using namespace std;
int main ()
{
```

```
 int x=5; // initial value: 5
 int y=2; // initial value: 2
 int z;
 z = x + y;
 cout << z;
 return 0;
}
```

Sometimes it is convenient to give a constant value to a variable. For example:

const pi= 3.1415926;

However if you declare pi without const also the program will work. Most first time programmers from observation do not use the const variable definition.

1.5.1 Introduction to Strings

While a character represents a single symbol such as ‘A’ or a ‘$’ a string represents a compound type in C++.

To use a string we need to include a new directive # include <string> in addition to the directive iostream.

We use cout to output the value of x to the screen.

```
// An example of a string
#include <iostream>
#include <string>
using namespace std;
int main ()
{
 string x;
 x = “This is an example of a string”;
 cout << x;
 return 0;
}
```

In the example above the output would be the string:

```
This is an example of a string
```

1.5.2 Operators

Operators are used on variables and constants. There are several types of operators.

The most common of the operators in C++ are:

- **Assignment operators (=):**

  x = 2; The assignment operation always takes place from right to left. So here, x is assigned the value of 2.

  y = z; The statement assigns the value contained in z to the variable y. The value currently in y is lost and replaced with the value in z. If z changes at a later moment, it will not affect the new value taken by y.

```
// assignment operator
#include <iostream>
using namespace std;
int main ()
{
 int x, y;
 x = 5; // x:5, y:
 y = 2; // x:5, y:2
 x = y; // x:2, y:2
 y = 3; // x:2, y:3
 cout << “x:”;
 cout << x;
 cout << “y:”;
 cout << y;
}
```

- **Arithmetic Operators (+, -, *, /, %):**

The five arithmetical operations supported by C++ are common to almost all languages and something one is familiar with from mathematics.

They are:

Operator	**Description**
+	Addition
-	Substraction
*	Multiplication
/	Division

%	Modulo

The modolu operator gives the remainder of a division of two values: For example:

x = 10 % 4; will give the value of 2 to x.

- **Compound Assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)**

Compound assignment operators modify the current value of a variable by performing an operation on it. For example:

a += x; means a= a + x;

x -=10; means x = x-10;

y /= x; means y = y / x;

z *= x; means z = z * x

- **Increment and Decrement (++, —)**

These operators increase or decrease by one (1) the value stored in a variable. This is equivalent to +=1 and - -1 respectively.

Thus ++y;

y+= 1;

y = y + 1;

are all equivalent in its functionality. They all increase by one the value of y.

So does it make a difference if the ++ is before (prefix) or after the variable (suffix)?

Yes it does:

Example 1	**Example 2**
z =5	z= 5;
y = ++z;	y = z++
Here z is 6, y is 6	Here z is 6 y is 5

In other words in Example 1, the value assigned to y is the value of z after being increased. In Example 2, the value assigned to y is the value of z before being increased.

- **Relational and comparison operators (==, !=, >, <, >=, <=)**

Two expressions can be compared using relational and equality operators. The result of such an operation is either true or false. The relational operators in C++ are:

Operator	Description
= =	Equal to
!=	Not equal to
<	Less than
>	Greater than
< =	Less than or equal to
>=	Greater than or equal to

Thus:

```
(7 == 8) is false
5 > 3 is true
1 != 2 is true
7 >= 7 is true
9 < = 8 is false
```

The assignment operator (=) is not the same as the relational operator (= =). One assigns the value on the right hand side to the variable on the left and the other compares whether values on both sides of the operator are equal.

- **Logical operators (!, &&, ||)**

The operator! is the C++ operator for the Boolean operation NOT. Thus:

```
! (7 = = 7) is false
! (7 < = 5) is true
! (9 > 5) is false
! (10 <=8) is true
```

The logical operators && and!! are used when evaluating two expression to obtain a single relational result. The && is equivalent to the Boolean logical operation AND which has the following truth table.

x	y	x && y
True	True	True

True	False	False
False	True	False
False	False	False

The || logical operator is equivalent to the Boolean logical operation OR which has the following truth table.

x	y	x || y
True	True	True
True	False	True
False	True	True
False	False	False

- **Conditional ternary operator (?)**

The conditional operator returns one value if the expression evaluates to true and a different one if the expression evaluates to false. Its syntax is:

condition ? result1: result2

This is similar to using the IF condition. We will discuss this in the controls chapter.

For now:

9 = = 7 ? 5: 3	will give the result 3 because 9 and 7 are not equal
7 = = 6 + 1 ? 5: 3	will give the result 5

Let us look at an example:

```
// conditional operator
#include <iostream>
using namespace std;
int main ()
{
 int x, y, z;
 x=1;
 y=5;
 z = (x < y) ? 5: x;
```

```
 cout << z << '\n';
}
```

In the code above the result would be 5 because 1 is less than 5. Had it been greater than y then the result would give the value of x.

- **Comma Operator (,)**

The comma operator separates two or more expressions. For example:

```
x = (y = 5, y +7);
```

This would assign the value of 5 to y and then assign y +7 to x. The result would x =9 while y = 5.

- **Bitwise operators (&, |, ^, ~, <<, >>)**

Bitwise operators modify variables considering the bit patterns that represent the values they store.

Operator	Equivalent	Description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement
<<	SHL	Shift bits left
>>	SHR	Shift bits right

- **Explicit type casting operator**

Type casting operators allow conversion of a value of a given type to another type. For example:

```
int x;
float y = 3.15;
x= (int) y;
```

Here the x will be 3. Another way of writing the last statement would be:

```
x = int (y);
```

Both ways are valid in C++.

- **Size of operator**

This operator accepts one parameter, which can be either a type or a variable and returns the size in bytes of that type or object.

y = sizeof (char);

Here y will get the value 1 because char is a type with a size of 1 byte. The value returned by sizeof is a compile time constant so it is always determined before program execution.

1.5.3 Precedence of operators

A single expression may have multiple operators. For example:

z = 7 + 8 / 2;

So where do one start?

Had this been written as: z = 7 + (8 / 2); it would have been easier as the bracket items will be evaluated first because of precedence. 7 + 4 = 11. The table below summarizes the precedence order.

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++, —, (),[]	Increment, decrement, bracket.	Left-to-right
3	Prefix (unary)	++,—,|!	prefix increment / decrement, not, logical not	Right-to-left
4	Pointer-to-member	.* ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or	|	bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right

14	Disjunction	||	logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= |=	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

1.5.4 Scope of a variable

We have looked into variable declaration. Variables can be local or global.

Variables declared inside a function or block are local variables. Only statements that are inside that function or block of code can use them. Local variables are not known to functions outside their own.

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the lifetime of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout the entire program.

We will look at two examples:

```
#include<iostream>
using namespace std;
int main ()
{
 int x, y;
 int z;
 x =10;
 y =20;
 z = x + y;
 cout << z;
 return 0;
}
```

```
#include<iostream>
using namespace std;
int t;
int main ()
{
 x =10;
 y =20;
 t = x + y;
 cout << t;
 return 0;
}
```

In the table on the left, there are only local variables. The output will be 30.

The second table has a declaration for t outside main function. The main function uses it to print the value of t, which remains at 30.

C++ allows for global variables and local variables to be the same- but it is best not to use them that way.

2

I / O

"The greatest sign of success for a teacher … is to be able to say, 'The students are now working as if I did not exist.'" — Maria Montessori

2.0 Input and Output

We will look at cin and cout here.

Stream	Description
cin	standard input
cout	standard output
cerr	standard error –output
clog	standard logging- ouput

2.1 cout

cout is used with << which is used for insertion.

Each cout line produces a line of words on a separate line only if the end line \n or endl is used.

Otherwise, the lines will be written one after the other on the same line.

```
cout << "Sentence 1"; // prints Sentence 1
cout << 100; // prints 100
cout << x; // prints the value of x
```

Text is enclosed in double quotes as shown in previous example. Multiple insertion operations (<<) may be chained in a single statement.

```
cout << "This is" << "a" << "a valid statement";
```

What cout will not do automatically is add line breaks at the end, unless instructed to do so. So if one writes the following lines of code:

```
cout << "This is line 1.";
```

```
cout << “This is line 2.”;
```

Then the output will be in a single line without any line breaks in between as shown below:

This is line 1. This is line 2.

To insert a line break, a new line character should be included as follows:

```
cout << “This is line 1.\n”;
cout << “This is line 2.\n”;
```

Now the output will be:

This is line1.

This is line 2.

Another way is to use endl. For example:

```
cout << “This is line 1.” << endl;
cout << “This is line 2.” << endl;
```

2.2 cin

The standard input by default is the keyboard and we will now look at cin as the stream object to access what is to be input.

```
int x;
cin >> x;
```

The value of x can be entered.

We will use both cin and cout in this example.

```
// example of cin and cout
#include <iostream>
using namespace std;
int main ()
{
 float x;
 cout << “Please enter a number: ”;
 cin >> x;
 cout << “The value you entered is” << x;
```

```
 cout << “and its double is” << x*2 << “.\n”;
 return 0;
}
```

Here the program asks for a number of type float (ex: 3.14) and gives its double. Therefore, if we entered 5.0 then the output would be 10.0. If you are running this program as a standalone, you may see the command window closing very fast. In this case enter cin.get(); after second and third cout statements.

Like cout, one can chain cin to request more than one data in a single line statement. For example:

cin >> x >> y;

is equivalent to:

cin >> x;

cin >> y;

Let us now look at an example where cin, cout is used to get a string from the keyboard. When writing a console-based program in C++, cin might not effectively capture the string. Therefore, we have to introduce a function called getline in the example below:

```
// cin, cout, getline
#include <iostream>
#include <string>
using namespace std;
int main ()
{
 string x;
 cout << “What’s your name?”;
 getline (cin, x);
 cout << “Hello” << x << “.\n”;
 return 0;
}
```

getline will call cin along with the string variable x. x stores the values of the question being asked in the program.

If you are using this is a console program you may have to write cin.get(); after the last cout statement to keep the command window from closing.

3

The flow control

"The important thing in life is not the triumph but the struggle."
— Pierre de Coubertin

3.0 Statements

By now, you are familiar with the general layout of a C++ program. Statements are the individual instructions of a program for example- variable declaration and expressions.

In a C++ program, a statement can be a simple one line statement terminated by a semicolon (;) or a compound statement.

Compound statement is a group of statements each terminated by a semicolon (;) and grouped together in a block enclosed in curly brackets {} as follows:

```
{
 float x;
 cout << "Please enter a number: ";
 cin >> x;
 cout << "The value you entered is" << x;
 cout << "and its double is" << x*2 << ".\n";
 return 0;
}
```

3.1 If and else

The *if* keyword is found in several programming languages. It is used to execute a statement or block if and only if a condition is fulfilled.

if (condition) statement

```
if (x < 100)
cout << "x is less than 100";
```

If the condition is not met, nothing is printed out. If there are multiple statements to be executed when the condition is fulfilled, then these statements should be enclosed in brackets forming a block.

For example:

```
if (x < 100)
```

```
{
 cout << “x is less than 100”;
 cout << x;
}
```

The code above can be made on a single line as follows:

```
if (x < 100)
{cout << “x is less than 100”; cout << x;}
```

Generally, beginners are more comfortable with writing a statement on its own line and this can increase the readability of the code. The *else* keyword is used to introduce an alternative statement. The syntax is:

If (condition) statement1 else statement2

For example:

```
if (y == 10)
cout << “y is equal to 10”;
else
cout << “y is not 10”;
```

So here the statement y is equal to 10 will print if (y == 10) otherwise the statement *y is not 10* will print.

3.2 Loops

Loops are known as iteration statements. That is because one uses loops to repeat a statement a certain number of times while a condition is fulfilled. The loop uses the keywords- while, do and for.

A ***while loop*** has the following syntax:

while (condition) statement

```
// example of while loop
#include <iostream>
using namespace std;
int main ()
{
 int y = 3;
 while (x>0) {
     cout << y << “, ”;
```

```
    —y;
 }
 cout << “End of loop\n”;
}
```

The output of this code would be 3, 2, 1, End of Loop. To control the command window use cin.get(); after the last cout if necessary. If after any execution of the statement the condition become false then the loop ends and the program continues right after the loop.

In this case, the condition was x had to be greater than zero. So when it counts from five downwards one at a time due to (—x) it will stop the loop after one when x will not be greater than zero. Then it jumps the loop and goes to the next available line in the program, which in this case is the cout statement. Thus, it will print “End of loop”.

In all loops, there must be an exit point. Otherwise, it will go on forever.

do-while loop

The syntax for this type of loop is:

do statement while (condition);

The condition is evaluated after the statement has been executed. Thus, at least one execution of the statement is done even if the condition is not fulfilled. For example:

```
// do-while loop
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string x;
 do {
     cout << “Enter your username: ”;
     getline (cin, x);
     cout << “You entered: ” << x << ‘\n’;
 } while (x != “student”);
}
```

Here let us look at the output based on the input:

```
Enter your username: hello
You entered: hello
Enter your username: student
You entered: student
```

So at least one time the loop has to be executed even if one enters student to exit the loop like in this case. This feature of do is the reason why one may select it over while.

The for loop

The syntax for this type of loop is as follows:

for (initialization; condition; increase/decrease) statement;

Like the while loop this type of loop repeats statements while condition is true. In addition the for loop provides an initialization and increase/decrease (change) expression executed before the loop begins for the first time and after each iteration.

for (int x=5; x > 0; x—)

x starts at five and is decremented by one each time as long as the value of x does not go below one.

The three fields in a for loop are optional. They can be left empty, but the semicolon cannot be omitted. For example:

for (;x<5;) is a loop with an initialization or an increase and is equivalent to a while loop.

for (;x<5;++x) is a loop with increase but no initialization. In this case the variable might have been initialized already elsewhere in the program.

for (x=5; ;++x) is a loop with no condition. Thus, it is the same as a loop with true as condition. Thus resulting an infinite loop or endless loop.

```
#include <iostream>
using namespace std;
int main ()
{
 for (int n=5; n>0; n—) {
 cout << n << “, ”;
}
 cout << “End of line\n”;}}
```

It produces the same result -5, 4,3,2,1, End of line.

3.3 Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations. There are three types of jump statements. They are break, continue and goto. Let us look at an example:

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration is if the end of the block has been reached causing it to jump to the start of the following iteration. The number 3 is skipped.

```
for (int x=5; x>0; x—) {
 if (x==3) continue;
 cout << x << “, ”;
}
 cout << “Number skipped\n”;}
```

The goto statement

Goto allows an absolute jump to another point in the program. It uses a label with a:

```
#include <iostream>
using namespace std;
int main ()
{
 int a = 10;
 LOOP:do
 {
     if(a == 15)
     {
     a = a + 1;
     goto LOOP;
     }
     cout << “value of a: ” << a << endl;
     a = a + 1;
 }while(a < 20);
 return 0;
}
```

Here the output would be a list of numbers from 10 to 19 without the number 15. Here we see a do while loop. The number counts from 10 to 19 but skips 15.

3.4 Switch

```
switch (expression)
{
 case constant1:
 group of statements -1;
 break;
```

```
 case constant2:
 group of statements-2;
 break;
 .
 default:
 default group of statements
}
```

Switch evaluates expression and checks if it is equivalent to constant 1. If it is, it executes the group of statements-1 until it reaches the break statement. At the break statement, the program jumps to the end of the entire switch statement – the closing brace.

If expression is not equal to constant1, it then checks against constant2 and if equal executes the group of statements-2 until a break is found and then jumps to the end of the switch statement.

Finally, if the value of the expression did not match any of the constants the program executes the statements included after the default: label if it exists.

Let us look at an example:

```
#include <iostream>
using namespace std;
int main ()
{
 char x = 'C';
 switch(x)
 {
    case 'A':
    cout << "Sedan" << endl;
    break;
    case 'B':
    cout << "Mini Van" << endl;
    break;
    case 'C':
    cout << "Truck" << endl;
    break;
    default:
    cout << "Invalid entry" << endl;
 }
 cout << "Your selection is" << x << endl;
 return 0;
}
```

The output will be: Truck

Your selection is C

If anything other than A, B, C is entered the default message will be displayed which is invalid entry.

4

Functions

"Every truth has four corners: as a teacher I give you one corner, and it is for you to find the other three." — Confucius

4.0 Definition of a function

A function is a set of statements that together perform a task.

A function has a function's name, return type, and parameters. There can be more than one function declaration in a C++ program.

The general syntax of a function is as follows:

```
return_type function_name(parameter list)
{
 body
}
```

- **Return Type**: The **return_type** is the data type of the value the function returns. If functions do no return, a value the return_type is set to void.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. A function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Let us look at an example:

```
// function example
#include <iostream>
using namespace std;
int multiply (int x, int y)
{
 int z;
 z = x * y;
 return z;
```

```
}
int main ()
{
 int w;
 w = multiply (5,2);
 cout << “The result is” << w;
}
```

Here is a function, *multiply* that has been defined outside the main function. Inside the main function we have called the function by its name *multiply* and passed to it two values 5 and 2. The values will be assigned to x =5 and y =2. The result will be 10.

Remember the function is called from main. At this point, activity is transferred to the function and once the function has been executed, control returns to the point in main where the function was called.

If one looks at the *multiply* function it returns a value z back to main which is of the same type as w. The value of w will be printed with the value of z computed in the function block.

A function can be called several times inside a main function.

4.1 Functions that do not return a value

Let us look at the following example:

```
// void function example
#include <iostream>
using namespace std;
void pmessage ()
{
 cout << “Hello I am function!”;
}
int main ()
{
 pmessage ();
}
```

In this example, we are calling a function from main. It will print the message *Hello I am a function!*

Unlike the previous example in section, 4.0 no values are transferred to the function and none are computed and returned.

While most functions use the call by value method as in the previous example, the use of

void term in defining function pmessage tells that the function does not return a value.

Further note there are no parameters in the bracket of the *pmessage* function. The absence of these parameters means that the function is not going to receive any values from the main function.

If the execution of main ends normally without encountering a return (return 0;) statement the compiler assumes the function, ends with an implicit return statement.

All other functions with a return type shall end with a proper return statement that includes a return value.

There can be many functions in a C++ program but the main() function must always be there irrespective of how complicated the program is.

4.2 Default values for parameters

When you define a function, you can specify a default value for each of the last parameters. This value is used if the corresponding argument is left blank when the function is called.

Let us look at the following example:

```
// default values in functions
#include <iostream>
using namespace std;
int divide (int x, int y=5)
{
 int z;
 z=x / y;
 return (z);
}
int main ()
{
 cout << divide (10) << '\n';
 cout << divide (30,6) << '\n';
 return 0;
}
```

Here in the first call divide (10) the values transferred to x is 10 and since there is nothing else transferred 10 will be divided by y = 5 resulting in the answer 2.

In the second call divide (30, 6), function parameters x will take the value of 30 and y whose value in the function is 5 will get the value 6. Thus, the result will be 5.

5

Arrays

"Read not to contradict and confute, nor to believe and take for granted… but to weigh and consider."—Francis Bacon

5.0 What are Arrays

An array, stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. The index value 0 represent the lowest address.

5.1 Simple Arrays

```
double amount [5] = {1000, 800, 900, 700, 600}
```

The number of the elements of the curly bracket {} must not be more than the number in [] brackets. This is a single or one-dimensional array.

If the number in [] brackets is missing then an array of just enough size to accommodate the elements will be created.

If we write amount [4] =600 this would mean that the 5th place or 4th index on the array will have the value of 600.

0	1	2	3	4
1000	800	900	700	600

Amount

So as shown 5 cells numbered 0 through 4.

The fifth position represented by index 4 will hold the value of 600 in a one-dimensional array.

5.2 Accessing an element of an array

Storing values is an array is only good if we can access them. To access elements of an array we can use a loop such as a- for loop.

```
// arrays example
#include <iostream>
```

```
using namespace std;
int amount [] = {16, 2, 77, 40, 12071};
int n;
int v=0;
int main ()
{
 for (n=0 ; n<5 ; ++n)
 {
    v= v + amount[n];
 }
 cout << v;
 return 0;
}
```

This will give the sum of the five numbers in the array, which will be 12206.

5.3 Multi-dimensional Arrays

The format for a multidimensional array is as follows:

type name[size1][size2]…[size n];

The simplest form of the multidimensional array is the two-dimensional array. To declare a two-dimensional integer array of size x, y, you would write something as follows:

type array name [x][y];

Where type can be any valid C++ data type and array name will be a valid C++ identifier.

int a [2] [3] is an array with two rows and three columns. For example:

int a [2] [3] = {

{0, 1, 2},

{3, 4, 5}

};

So here, the row 1 has the index 0 or in simple words, we count rows starting with 0 and not 1. Thus the row number for the above array is 0 & 1 and column numbers are 0 … 2.

	Column 0	Column 1	Column 2
Row 0	0	1	2
Row 1	3	4	5

Now we access the elements of a multi-dimensional array in the same manner as we did for a single array by using a loop. Let us consider a four row two column array defined by

a [4][2]. Let us assign to it the following values:

a [4] [2] = {{0,0}, {1,2}, {2,4}, {3,6}}

To print out the values of this and similar arrays we will use the ‘for’ loop. The key loop would be:

```
for (int i = 0; i < 4; i++)
for (int j = 0; j < 2; j++)
{
 cout << “a[” << i << “][” << j << “]: ”;
 cout << a[i][j]<< endl;
}
```

Here we have i < 4 because we know the columns are numbered 0 through 3.

Here is the full code:

```
#include <iostream>
using namespace std;
int main ()
{
 // an array with 3 rows and 2 columns.
 int a[3][2] = {{0,0}, {1,2}, {2,4}};
 // output each array element’s value
 for (int i = 0; i < 3; i++)
 for (int j = 0; j < 2; j++)
 {
     cout << “a[” << i << “][” << j << “]: ”;
     cout << a[i][j]<< endl;
 }
 return 0;
}
```

The output would be similar to this:

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

6

Pointers

"Teaching students to count is fine,
but teaching them what counts is best."
— Bob Talber

6.0 What are Pointers

A pointer is a variable whose value is the address of another variable. Let us look at the following example:

```
#include <iostream>
using namespace std;
int main ()
{
int x;
cout << "Address of x variable: ";
cout << &x << endl;
return 0;
}
```

The output of this code will be similar to this:

Address of x variable: 0031FDE0

Address of y variable: 0031FDCC

What we see above is the memory location for variables x. Notice that the '&' is used to give the memory location.

This value will vary and may be different on your computer. However, what is important to understand to keep in mind is that they are memory addresses.

We can define pointers by first indicating its type as follows:

int *x;

float *y;

double *p;

char *w

These pointers will point to an integer, float, double and char respectively.

We can store the value of variable in a pointer and the address of the variable. Let us look

at the following example:

```
#include <iostream>
using namespace std;
int main ()
{
 int x = 10;
 int *ip; // pointer variable
 ip = &x; // store address of var in pointer variable
 cout << “Value of x variable: ”;
 cout << x << endl;
 cout << “Address stored in ip variable: ”;
 cout << ip << endl;
 cout << “Value of *ip variable: ”;
 cout << *ip << endl;
 return 0;
}
```

The output will be similar to this:

Value of x variable: 10

Address stored in ip variable: 002BFC90

*Value of *ip variable: 10*

6.1 Pointer concepts

Let us look at some pointer concepts:

- **Null pointers**

```
#include <iostream>
using namespace std;
int main ()
{
 int *p = NULL;
 cout << “The value of p is ” << p ;
 return 0;
}
```

This is an example of a NULL pointer. The output of this code is:

The value of p is 0

- **Pointer Arithmetic**

A pointer is an address, which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. Four arithmetic operators that can be used on pointers: ++, —, +, and –.

For the following example, we will define an array y of three elements. It is a single dimension array. We will use a pointer y to point to the array y. Let us assume both the pointer and the array are of type integer.

We will then a loop to go through each element of the array y and print out its memory address and value. We prefer using a pointer in the program instead of an array because the variable pointer can be incremented, unlike the array name, which cannot be incremented because it is a constant pointer.

A for loop is used in this example to go through each element of the array y as was done in the chapter on arrays.

Let us the look at the example:

```
#include <iostream>
using namespace std;
const int x = 3;
int main ()
{
 int y[x] = {100, 200, 300};
 int *p;
 // let us have array address in pointer.
 p = y;
 for (int i = 0; i < x ; i++)
 {
     cout << “Address of y [” << i << “] = ”;
     cout << p << endl;
     cout << “Value of y [” << i << “] = ”;
     cout << *p << endl;
     p++;
 }
 return 0;
}
```

The expected output of the code is similar to:

Address of y[0] = 0031FD88

Value of y[0] = 100

Address of y[1] = 0031FD8C

Value of y[1] = 200

Address of y[2] = 0031FD90

Value of y[2] = 300

- **Pointer versus arrays**

Pointers and arrays are strongly related. A pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Let us look at the following:

```
#include <iostream>
using namespace std;
const int x = 3;
int main ()
{
 int y[x] = {100, 200, 300};
 int *p;
 // let us have array address in pointer.
 p = y;
 for (int i = 0; i < x; i++)
 {
     cout << “Address of y[” << i << “] = ”;
     cout << p << endl;
     cout << “Value of y[” << i << “] = ”;
     cout << *p << endl;
     // point to the next location
     p++;
 }
 return 0;
}
```

The output of the program would be similar to:

Address of y[0] = 004DF750

Value of y[0] = 100

Address of y[1] = 004DF754

Value of y[1] = 200

Address of y[2] = 004DF758

Value of y[2] = 300

If we use the array name y and write this y++ that is syntactically wrong.

7

Strings

"Education is not to reform students or amuse them or to make them expert technicians. It is to unsettle their minds, widen their horizons, inflame their intellects, teach them to think straight, if possible." — Robert M. Hutchins

7.0 Character String

We are familiar with the char data type. We also have gone through arrays and pointers in previous chapters.

Let us therefore look at the following code where we define a character array to print the word Yes.

```
#include <iostream>
using namespace std;
int main ()
{
 char x[3] = {'Y', 'e', 's', '\0'};
 cout << x << endl;
 return 0;
}
```

The expected output of this program is:

Yes

Here the declaration char x [3] is a one-dimensional array of characters terminated by a **null** character '\0'. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array.

7.1 String Functions

Some of the common string functions in C++ are:

No	Function Name	Description
1	strcpy (s1, s2)	Copies string s2 into string s1
2	strcat(s1, s2)	Concatenates string s2 onto the end of string 1
3	strlen(s1)	Returns the length of string s1
4	strcmp(s1, s2)	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1 > s2.

```
include <iostream>
#include <string>
using namespace std;
int main ()
{
 char x[10] = “Hello”;
 char y [10] = “World”;
 strcat(x, y);
 cout << “strcat(x, y): ” << x << endl;
 return 0;}
```

The output will be *Hello World*. Just keep an eye on the size of the arrays x and y.

8

References

"Education…is painful, continual and difficult work
to be done in kindness, by watching, by warning,…
by praise, but above all — by example."
— John Ruskin

8.0 What are References

A reference variable is an alias, that is, another name for an already existing variable. There cannot be NULL references and references must be to valid variables. References initialized to an object cannot refer to another object. References must be initialized when they are created.

```
include <iostream>
using namespace std;
int main ()
{
 int x;
 // declare reference variable
 int& y = x;
 x = 10;
 cout << "Value of x: " << x << endl;
 cout << "Value of y reference: " << y << endl;
 return 0;}
```

Expected output of this example is:

Value of x: 10

Value of y reference: 10

9

File

“Tell me and I forget. Teach me and I remember.
Involve me and I learn.” — Benjamin Franklin

9.0 Input and output with files

So far, we have used cin and cout as input and output for the programs we have written. Now look us look at how to deal with external files. C++ provides the following classes to achieve this. It requires the use of a new additional header other than iostream. The header is fstream.

Stream	**Description**
ofstream	To write to files
ifstream	To read from files
fstream	To read and write from and into files

Let us look at an example:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 ofstream myfile;
 myfile.open (“first.txt”);
 myfile << “Hello! I am writing to a file.\n”;
 myfile.close();
 return 0;}
```

This writes to a text file known as first.txt. The words Hello! I am writing to a file. If the file first.txt does not exist it will created as long as permissions on your computer allow for it. Otherwise, create the file first noting the directory of C++ compiler.

9.1 Opening a file

In order to open a file with a stream object we use the following syntax:

open (filename, mode);

Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

Flags	Description
ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file first.bin in binary mode to add data we could do it by the following call to member function open:

ofstream myfile;

myfile.open (“first.bin”, ios::out | ios:: app | ios:: binary);

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

Stream	Default Mode
ofstream	ios:: out
ifstream	ios:: in
Fstream	ios:: in	ios:: out

To check if a file stream was successful in opening a file, one can use the is_open member. The syntax would look like this:

if (first.is open()) {statements}

9.2 Closing a file

When we are finished with writing to a file or reading from it we have to close the file.

The syntax for that is as follows:

first.close();

9.3 Text Files

Text files are those where the ios:: binary flag is not included in their opening mode. They have the extentions .dat, .txt, .csv etc.

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 ofstream myfile (“first.txt”);
 if (myfile.is_open())
 {
     myfile << “This is a line.\n”;
     myfile << “This is another line.\n”;
     myfile.close();
 } else cout << “Unable to open file”;
 return 0;}}
```

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () {
 string line;
 ifstream myfile (“first.txt”);
 if (myfile.is_open())
 {
     while (getline (myfile, line))
     {
     cout << line << ‘\n’;
     }
     myfile.close();
 }
     else cout << “Unable to open file”;
     return 0;}}}
```

This program allows one to read from a text file.

10

Storage Classes

“Always to see the general in the particular is the very foundation of genius.” — Arthur Schopenhauer

10.0 Types of storage classes

Let us look at five storage classes:

- auto
- register
- static
- extern
- mutable

The auto storage class is the default storage class for all local variables.

```
{
 int y;
 autoint x;
}
```

The auto storage class can only be used within functions.The register storage class is used to define local variables that should be stored in the register rather than RAM.

```
{
 registerint x;
}
```

The register should only be used for variables that require quick access such as counters.

The **static** storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope.

Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may be applied to global variables

```
#include<iostream>
using namepace std;
void y(void);
```

```
static int count =5;/* Global variable */
int main()
{
 while(count—)
 {
     y();
 }
 return 0;
}
// Function definition
void y(void)
{
 static int x =0;// local static variable
 x++;
 std::cout << “x is” << x ;
 std::cout << “and count is” << count << std::endl;
}
```

The expected output for the above example is as follows:

x is 1and count is 4

x is 2and count is 3

x is 3and count is 3

x is 4and count is 1

x is 5and count is 0

The **extern** storage class gives a reference of a global variable that is visible to all the program files. When this class is used variables cannot be initialized.

```
#include<iostream>
int count;
extern void w();
main()
{
 count =5;
 w();
}
```

Let us assume the code above is in a file call first.cpp. Let also assume there is a second file called second.cpp

```
#include<iostream>
Extern int count;
void w(void)
{
 std::cout << “Count is” << count << std::endl;
}
```

Since the function was declared, with extern it is available to the second file and it is able to print the value of count as 5.The **mutable** member can be modified by a const member function.

11

Classes

"There are two kinds of teachers: the kind that fills you with so much quail shot that you can't move, and the kind that just gives you a little prod behind and you jump to the skies." — Robert Frost

11.0 Building classes

Classes form one of the fundamental features that enable C++ to be an object oriented language.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. Either a semicolon or a list of declarations must follow a class definition.

Let us look at an example of a class.

```
class Car
{
 public:
 double length // Length of the car
 double width; // Width of the car
 double weight; // Weight of the car
 double engine; // Engine size
};
```

We have defined a class known as car with a few features attributed to the car. The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object.

Now let us look at objects.

Let us assume there are two cars – car1 and car2. We will declare it as follows:

Car car1

Car car2

Both objects will now have a copy of the data members of the class known as Car. For example:

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

```
class Car
{
 public:
 double length; // Length of the car
 double width; // Width of the car
 double weight; // Weight of the car
 double engine; // Engine size
};
int main()
{
 Car car1; // Declare car1 of type Car
 Car car2; // Declare car2 of type Car
 car1.weight = 1500;
 car1.engine = 1400;
 car2.weight = 2000;
 car2.engine = 1800;
 cout << “Weight of car1: ” << car1.weight <<endl;
 cout << “Weight of car2: ” << car2.weight <<endl;
 return 0;
}
```

The output of the code above would be:

Weight of car1: 1500

Weight of car2: 2000

11.1 Class concepts

- **Class member function**

A member function of a class is a function that has its definition within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object. Let us look at an example:

```
class Marks
{
 public:
 double prelim; // narks in first exam
 double midterm; // marks in second exam
 double final; // marks in final exam
 double getfinal(void)
 {
    return (0.3 * prelim) + (0.3 * midterm) + (0.4 * final);
```

```
 }
};
```

Here the function getfinal() has been defined within the class. If the function is not defined within the class then use scope resolution operator,**::**, as follows:

- **Class modifiers**

Data hiding is one of the important features of Object Oriented Programming (OOP). The access labels - public, private, and protected, specify the access restriction to the class members.

A **public** member is accessible from anywhere outside the class but within a program. A **private** member variable or function cannot be accessed, from outside the class. Only the class and friend functions can access private members. Let us look at this:

```
#include <iostream>
using namespace std;
class Car
{
 public:
 double length;
 void setw(double w);
 double getw(void);
 private:
 double weight;
};
 // Member functions definitions
 double Car::getw(void)
{
 return weight ;
}
void Car::setw(double w)
{
 weight = w;
}
int main()
{
 Car car1;
 car1.length = 1100; // OK: because length is public
 cout << “Length of car: ” << car1.length <<endl;
 // car1.weight = 1500; // Error: because weight is private
 car1.setw(1500); // Use member function to set it.
```

```
cout << “Weight of car: ” << car1.getw() <<endl;
return 0;}
```

Here the declaration of weight as being private resulted in car1, which is an object of Car from being able to access the weight directly. The output of this program would be:

Length of car: 1100

Weight of car: 1500

A **protected** member variable or function is very similar to a private member. They can be accessed in child classes, which are called derived classes.

```
#include<iostream>
using namespace std;
class Car
{
 protected:
 double weight;
};
class Smallcar:Car // Smallcar is the derived class.
{
 public:
 void setSmallw(double wid);
 double getSmallw(void);
};
double Smallcar::getSmallw(void)
{
 return weight;
}
void Smallcar::setSmallw(double w)
{
 weight = w;
}
int main()
{
 Smallcar car1;
 car1.setSmallw(1500);
 cout << “Weight of car: “<< car1.getSmallw() << endl;
 return 0;
}
```

The output of the program would be:

Weight of car: 1500

- **Constuctors**

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Let us look at the following:

```
#include <iostream>
using namespace std;
class Car
{
 public:
 void setw(double w);
 double getw(void);
 Car(); // This is the constructor
 private:
 double weight;
};
 // Member functions definitions including constructor
 Car::Car(void)
{
 cout << “Please wait! Answer is on the way” << endl;
}
 double Car::getw(void)
{
 return weight;
}
 void Car::setw(double w)
{
 weight = w;
}
int main()
{
 Car car1;
 car1.setw(1500);
 cout << “Weight of car: ” << car1.getw() <<endl;
 return 0;}
```

Here the output would be the weight of the car. The difference is:

Please wait! Answer is on the way

Weight of car: 1500

- **Destructors**

```
#include <iostream>
using namespace std;
class Car
{
 public:
 void setw(double w);
 double getw(void);
 Car(); // This is the constructor declaration
 ~Car(); // This is the destructor: declaration
 private:
 double weight;
};
// Member functions definitions including constructor
Car::Car(void)
{
 cout << “Object is being created” << endl;
}
Car::~Car(void)
{
 cout << “Object is being deleted” << endl;
}
double Car::getw(void)
{
 return weight;
}
void Car::setw(double w)
{
 weight = w;
}
int main()
{
Car car1;
car1.setw(1500);
cout << “Weight of the vehicle: ” << car1.getw() <<endl;
return 0;
}
```

The expected output is:

Object is being deleted

The weight of the vehicle: 1500

12

Inheritance

“Stay committed to your decisions; but stay flexible in your approach.” —Tony Robbins

12.0 The concept of Inheritance

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class (

Let us look at an example by first writing a base class. Here the base class that we are going to define is called vehicle. The vehicle has certain properties such as length, width, weight, engine etc.

We will then create a derived class called car. As a car is a vehicle and it will inherit the properties of the base class that is vehicle.

The example is as follows:

```
#include <iostream>
using namespace std;
// Base class
class vehicle
{
 public:
 void setengine(double e)
 {
     engine = e;
 }
 public: double getengine()
 {
     return (engine);
 }
 protected:
 double weight;
 double engine;
};
// Derived class
class car: public vehicle
{
};
```

```
int main(void)
{
 vehicle car;
 car.setengine(1500);
 cout << “Engine: ” << car.getengine() << endl;
 return 0;
}
```

The output is 1500. Inheritance is a key element as to why C++ is an object-oriented language.

12.1 Some OOP’s terms

Polymorphism is a term in C++. This feature means that there are a hierarchy of classes, which are related to each other through inheritance. Polymorphism can result in a call to a function being replied to by another function depending on the object that called the function.

Placing a virtual function in the base class will deal with polymorphism.

Data abstraction means showing only necessary information and hiding any background details. Encapsulation is a feature that keeps data and functions that operate on the data safer from external misuse.

Any C++ program, which has public and private members, is an example of data encapsulation and data abstraction. For example:

```
class Car
{
 public:
 double getw(void)
 {
     return weight;
 }
 private:
 double weight;
}
```

13

Exceptions

“Always to see the general in the particular
is the very foundation of genius.” — Arthur Schopenhauer

13.0 Error handling

An exception is defined as a problem that happens when a program is running. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

```
try
{
 // protected code
}catch(ExceptionName a1)
{
 // catch block
}catch(ExceptionName a2)
{
 // catch block
}catch(ExceptionName an)
{
 // catch block
}
```

Exceptions can be introduced anywhere a bunch of statements are being executed. For example:

```
#include <iostream>
using namespace std;
double division(int v, int w)
{
 if(w == 0)
 {
     throw “Division by zero”;
 }
 return (v/w);
}
int main ()
```

```
{
 int x = 10;
 int y = 0;
 double z = 0;
 try {
     z = division(x, y);
     cout << z << endl;
 }catch (const char* msg) {
     cerr << msg << endl;
     }
 return 0;
}
```

14

Conclusion

The C++ language is a very versatile programming tool.

The previous chapters concentrated on creating the basic building blocks of a standalone C++ program. Chapters 11 and 12 introduced topics important to object oriented programming. As you go on to more advanced programming you will encounter other OOP's concepts such as polymorphism, encapsulation, data abstraction and data hiding in detail.

Interestingly in the examples we have done, some of these concepts are already present.

From this point, one could take a more advanced look at C++ programming keeping in mind that wide usage of this language.

If you need, the zip file of all the programs in the chapters that we have done or updates on this subject and subsequent and related topics visit http://beam.to/fjtbooks