UNIT ONE

Introduction

Computer Program

Computers are so widespread in our society because they have three advantages over us humans.

- First, computers can store huge amounts of information.
- Second, they can recall that information quickly and accurately.
- Third, computers can perform calculations with lightning speed and perfect accuracy.

The advantages that computers have over us even extend to thinking sports like chess.

However, we have one very significant advantage over computers. We think on our own, while computers don't, at least not yet anyway. Indeed, computers fundamentally are far more powerful than brain. A computer cannot do anything without step-by-step instructions from you as a programmer telling it what to do. These instructions are called a computer program, and of course are written by a human, namely a computer programmer. Computer programs enable us to invest the computer's huge power.

Programming Language

Despite the huge power of computers, they still understand very primitive language, a language with two options: on (one) or off (zero). The language that computers understand is called machine language.

While computers think in ones and zeroes, the humans who write computer programs usually don't. Additionally, a complex program may consist of thousands or even millions of step-by-step machine language instructions, which would require a long amount of time to write. This is an important consideration since the amount of time within which a program has to be written is becoming increasingly less and less.

Fortunately, we do not have to write instructions to computers in machine language. Instead, we can write instructions in a programming language. Programming languages are far more understandable to programmers than machine language because programming languages resemble the structure and syntax of human language, not ones and zeroes. Additionally, code can be written much faster with programming languages than machine language because programming languages automate instructions; one programming language instruction can cover many machine language instructions.

C++ is one of many programming languages. Other popular programming languages include Java, C#, and Visual Basic. There are many others. Indeed, new languages are being created all

the time. However, all programming languages have essentially the same purpose, which is to enable a human programmer to give instructions to a computer.

Why learn C++ instead of another programming language? First, it is very widely used, both in industry and in education. Second, many other programming languages, including Java and C#, are based on C++. Indeed, the Java programming language was written using C++. Therefore, knowing C++ makes learning other programming languages easier.

Although programmer are not ought to write programs in machine language, the computers understands only machine language, this means that programmers write in high level language to computers that understand machine language, this raises the need of an interface or translator, this is what is called compiler

Compiler

The compiler is another program that translates the preprocessed source code (the source code after the insertions made by the preprocessor) into corresponding machine language instructions, which are stored in a separate file, called an object file, having an .obj extension. There are different compilers for different programming languages, but the purpose of the compiler is essentially the same, the translation of a programming language into machine language, no matter which programming language is involved.

The compiler can understand your code and translate it into machine language only if your code is in the proper syntax for that programming language. C++, like other programming languages, and indeed most human languages, has rules for the spelling of words and for the grammar of statements. If there is a syntax error, then the compiler cannot translate your code into machine language instructions, and instead will call your attention to the syntax errors. Thus, in a sense, the compiler acts as a spell checker and grammar checker.

Your First C++ Program

Before getting into any theory, let's look at a simple C++ program.
We will start by entering, compiling, and running the following program.
Now, go to your compiler, and enter the following code

```
/* The very first C++ program
Enter this program, then compile and run it.
*/
#include <iostream.h>
// main() is where program execution begin
void main()
```

```
{
cout << "This is my first C++ program";
}
```

You will follow these steps.
1. Enter the program.
2. Compile the program.
3. Execute the program.

Before beginning, it is necessary to define two terms. The first is *source code.* Source code is the version of your program that humans can read. The preceding listing is an example of source code. The executable version of your program is called *object code* or *executable code.* Object code is created by the compiler when it compiles your program.

Entering the Program

Enter the programs by hand. Typing in the programs yourself often helps you remember the key concepts.

Compiling the Program

Running the Program

A Line-by-Line Explanation

Now that you have successfully compiled and run the first sample program it is time to understand how it works. Towards this end, we will examine the program line by line. The program begins with the lines

```
/* The very first C++ program.
Enter this program, then compile and run it.
*/
```

This is a *comment.* Like most other programming languages, C++ lets you enter remarks into a program's source code. The contents of a comment are ignored by the compiler.

The purpose of a comment is to describe or explain the operation of a program to anyone reading its source code. In the case of this comment, it identifies the program.
In more complex programs, you will use comments to help explain what each feature of the program is for and how it goes about doing its work. In other words, you can use comments to provide a description of what your program does.
In C++, there are two types of comments.

- The one you've just seen is called a *multiline comment.* This type of comment begins with a **/*** (a slash followed by an asterisk). It ends only when a ***/** is encountered. Anything between these two comment symbols is completely ignored by the compiler. Multiline comments may be one or more lines long.
- The second type of comment is found a little further on in the program; we'll be discussing it shortly.

The next line of code looks like this:

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

The C++ language defines several *headers*, which contain information that is either necessary or useful to your program. For this program, the header **<iostream.h>** is needed. (It is used to support the C++ I/O system.) This header is provided with your compiler.
A header is included in your program by using the **#include** directive. Later in this book, you will learn more about headers and why they are important.

The next line in the program is

```
// main() is where program execution begins.
```

This line shows you the second type of comments available in C++: the *single-line comment.*
Single-line comments begin with // and stop at the end of the line.

The next line, as the preceding comment indicates, is where program execution begins:

```
void main()
```

The **main ( ) is a function, and it is t**he only function that any C++ program *must* contain,
main() function is where program execution begins and (most commonly) ends. (Technically speaking, a C++ program begins with a call to **main( )** and, in most cases, ends when **main( )** ends

The **void** that precedes **main( )** specifies that main() will return nothing to the system, this is will be explained further in Functions Chapter.

The next line in the program is the opening curly brace { on the line that follows **main( )** marks the start of the **main( )** function's code.

The next line in the program

```
cout << "This is my first C++ program.\n";
```

This is a console output statement. It causes the message

This is my first C++ program.

to be displayed on the screen. And a new line will be added because of **\n**.

\n is called escape sequence, in the next section it will be discussed more thoroughly

The << operator causes whatever expression is on its right side to be displayed on the output screen.

cout is a predefined identifier that stands for *console output*.

Thus, this statement causes the message to be output to the screen.

Notice that this statement ends with a semicolon.

In fact, all C++ statements end with a semicolon.

The last line in the program is

```
}
```

This line is the end of **main ( ), it** concludes the program.

A C++ program ending when the closing curly brace of **main ( )** is executed.

Handling Syntax Errors

It is easy to accidentally type something incorrectly when entering code into your computer.

Fortunately, if you enter something incorrectly into your program, the compiler will report a *syntax error* message when it tries to compile it.

But you have to be careful when handling errors, because the reported error may not always reflect the actual cause of the problem.

In the preceding program, for example, an accidental omission of the opening curly brace after **main( )** will cause some compilers to report the **cout** statement as the source of a syntax error.

Therefore, when you receive a syntax error message, be prepared to look at the two or three lines of code that precede the point at which the error is flagged.

Also you have to be careful about error numbers, sometimes correcting an error may uncover the existence of other errors, so when handling errors be sure of what you are doing and do not be suspicious.

Many errors in C++ are caused because of the fact that C++ is case-sensitive language, i.e main() is correct, but Main() is totally is different and causes an error if used instead of main().

I/O in iostream

In the previous program we introduced **cout** , which deals with output, and we introduced the escape sequence concept, namely **\n**, next is a listing of common escape sequences

Common Escape Sequences

Escape Sequence	**Name**	**What It does**
\a	Alarm	Causes the computer to beep
\n	newline	Causes the cursor to go to the next line
\t	Tab	Causes the cursor to go to the next tab stop
\\	Backslash	Causes a backslash to be printed
\'	Single quote	Causes a single quote to be printed
\”	Double quote	Causes a single quote to be printed

That is about output, concerning input, cin>> is used to take input from the user via keyboard, but before using cin>>, a memory location must be reserved to receive data into it. Following we will discuss variables.

Unit 2

Variables:

- A variable is a memory location to hold data
- A variable is a named memory location to store data;
- The content of a variable is changeable, not fixed.
- You need to reserve memory before you can store information there.
- You reserve memory by *declaring* a variable.
- Declaring a variable not only reserves memory, but also gives you an easy way of referring to that reserved memory when you need to do so in your program

Declaring Variables Syntax

You have to *declare* a variable before you can use it. Declaring a variable involves the following syntax:

```
datatype variablename ;
```

The data type may be any of the following:

Table 1 Data Types

Data Types	**Values stored**
int	This value is for whole numbers, or integers. The size depends on the operating systems. In 32-bit operating systems, the int is usually 32 bits (4 bytes) in length.
long	This data type holds larger whole numbers
float	Floats are used to hold decimal numbers such as 2.02798
double	A double is simply a really big float.
char	A char can store a single alpha-numeric type. In other words, it can store a number or a letter.
bool	Bool is short for Boolean, a reference to Boolean logic. Therefore, this data type can only store true or false.
short	This is basically a small integer. Usually 2 bytes in length. But the actually size will depend on the operating system. On 32-bit operating systems such as like *Windows 98, 2000, NT*, and *XP*, a short will be 2 bytes long.

The data type tells the computer how much memory to reserve, and what type of data will be saved.

Then you choose the variable name, according to the following:

Variables have names used to identify them to be referred to in code. There are only a few conditions on how you can name a variable.

- The variable name cannot begin with a number, only starts with a letter of the alphabet (A–Z or a–z) or an underscore (_). However, the second and following characters of the variable name may be digits, letters, or underscores.
- The variable name cannot contain spaces.
- The variable name cannot contain punctuation marks other than the underscore character (_).
- The variable name cannot be the same as a reserved word by C++, such as main or float.
- The variable name cannot have the same name as the name of another variable declared in the same place (scope). For present purposes, this rule means you cannot declare two variables in main with the same name.

It is a good idea to give your variables names that are meaningful

Finally, the variable declaration ends with a semicolon. The semicolon tells the compiler that the statement has ended, here are some examples:

int Counter;
float Average;

If you refer to a variable before declaring it you will receive a compiler error

A Second Simple Program

The following program creates a variable called **x,** gives it the value 1023, and then displays the message **This program prints the value of x: 1023** on the screen.

```
// Using a variable
#include <iostream.h>
void main()
{
int x; // this declares a variable
x = 1023; // this assigns 1023 to x
cout << "This program prints the value of x: ";
cout << x; // This displays 1023
}
```

This program introduces two new concepts.
First, the statement declares a variable called **x** of type integer

```
int x; // this declares a variable
```

In C++, all variables must be declared before they are used.
Further, the type of values that the variable can hold must also be specified. This is called the *type* of the variable. In this case, **x** may hold integer values. These are whole-number values whose range will be at least –32,768 to 32,767.
In C++, to declare a variable to be of type integer, precede its name with the keyword **int**.

The second new feature is found in the next line of code:

```
x = 1023; // this assigns 1023 to x
```

As the comment suggests, this assigns the value 1023 to **x**.
In C++, the assignment operator is the single equal sign. It copies the value on its right side into the variable on its left. After the assignment, the variable **x** will contain the number 1023.
The two **cout** statements display the output generated by the program.
Notice how the following statement is used to display the value of **x**:

```
cout << x; // This displays 1023
```

In general, if you want to display the value of a variable, simply put it on the right side of << in a **cout** statement.
In this case, because **x** contains the number 1023, it is this number that is displayed on the screen.

A More Practical Example

The next program actually performs a task: It converts miles to kilometers. It also shows how to input information.

```
// This program converts miles to kilometers.
#include <iostream.h>
void main()
{
double miles, kilometers;
cout << "Enter number of miles: ";
cin >> miles; // this inputs from the user
kilometers = miles * 1.6; // convert to kilometers
cout << "Kilometers: " << kilometers<<endl;
}
```

This program first displays a message on the screen saying

```
Enter number of miles:
```

and then waits for user (you) to enter a number amount of miles.
The program then displays the kilometer equivalent value.
For example, if you enter 2.5 miles, the program responds with the metric equivalent of 4 kilometers.

```
Enter number of miles: 2.5
Kilometers: 4
Press any key to continue
```

The first new thing you see in this program is that you can declare two variables in the same statement in the form of a comma-separated list.
In general, you can declare any number of variables of the same type by separating them with commas. This will be discussed later on.
Another new thing is using **endl** to insert a new line.

Declaring Multiple Variables of the Same Data Type

If you have several variables of the same data type, you could declare each variable in a separate statement.

```
int testScore;
int myWeight;
int myHeight;
```

However, if the variables are of the same data type, you don't need to declare each variable in a separate statement. Instead, you can declare them all in one statement, separated by commas. The following one statement declares all three integer variables:

```
int testScore, myWeight, myHeight;
```

Assigning Values to Variables

The purpose of a variable is to store information. Therefore, after you have created a variable, the next logical step is to specify the information that the variable will store. This is called *assigning* a value to a variable.

A variable can be assigned a value supplied by the programmer in code. A variable also can be assigned a value by the user, usually via the keyboard, when the program is running.

You may use the assignment operator to specify the value to be stored in a variable.

Assignment Operator

You use the assignment operator to assign a value to a variable. The syntax is

```
VvariableName = value;
```

The assignment operator looks like the equal sign. However, in C++ the = sign is not used to test for equality; it is used for assignment, in C++ the equal sign is = =, also called the equality operator.

The variable must be declared either before, or at the same time, you assign it a value, not after that. In the following example, the first statement declares the variable, and the second statement assigns a value to that variable:

```
int testScore;
testScore = 95;
```

The next example concerns *initialization*, which is when you assign a value to a variable as part of the same statement that declares that variable:

```
int testScore = 95;
```

However, the variable cannot be declared after you assign it a value. The following code will cause the compiler error "undeclared identifier" at the line testScore = 95:

```
testScore = 95;
int testScore;
```

As mentioned earlier in the "Declaring Variables" section, this compiler error will occur even though the variable is declared in the very next line because the compiler reads the code from top to bottom, so when it reaches the line testScore = 95, it has not seen the variable declaration.

The value assigned need not be a literal value, such as 95. The following code assigns to one integer variable the value of another integer variable.

```
int a, b;
a = 44;
b = a;
```

The assignment takes place in two steps:

- First, the value 44 is assigned to the variable *a*.
- Second, the value of *a*, which now is 44, is assigned to the variable *b*.

You also can assign a value to several variables at once. The following code assigns 0 to three integer variables:

```
int a, b, c;
a = b = c = 0;
```

The assignment takes place in three steps, from right to left:

1. The value 0 is assigned to the variable *c*.
2. The value of the variable *c*, which now is 0, is next assigned to the variable *b*.
3. The value of the variable *b*, which now is 0, is assigned to the variable *a*.

Finally, you can assign a value to a variable after it has already been assigned a value. The word "variable" means likely to change or vary. What may change or vary is the variable's value. The following code demonstrates a change in the value of a variable that was previously assigned a value:

```
#include <iostream.h>
void main(void)
{
   int testScore;
   testScore = 95;
   cout << "Your test score is " << testScore << "\n";
   testScore = 75;
   cout << "Your test score now is " << testScore << "\n";

 }
```

The output is

```
Your test score is 95
Your test score now is 75
```

```
The assignment operator is binary operator, it has two operands, left
and right, the left operand must be always a variable, not a literal
value, nor a constant, neither an expression, only a variable is
allowed on the left side of an assignment operator. For example:
 int x=9,b=81;
x=b=19;         // no problem
x=b+10;         // no problem
x=b+2=9;        // error
x*5=b;          // error
```

Assigning a "Compatible" Data Type

The value assigned to a variable must be compatible with the data type of the variable that is the target of the assignment statement. Compatibility means, generally, that if the variable that is the target of the assignment statement has a numeric data type, then the value being assigned must also be a number.

The following code is an example of incompatibility. If it is placed in a program, it will cause a compiler error.

```
   int testScore;
   testScore = "Jeff";
```

The description of the compiler error is “cannot convert from ‘const char [5]’ to ‘int’.” This is the compiler’s way of telling you that you are trying to assign a string to an integer, which of course won’t work; “Jeff” cannot represent an integer.

The value being assigned need not necessarily be the exact same data type as the variable to which the value is being assigned. In the following code, a floating-point value, 77.83, is being assigned to an integer variable, *testScore.* The resulting output is “The test score is 77.”

```
#include <iostream.h>
void main(void)
{
   int testScore;
   testScore = 77.83;
   cout << "The test score is " << testScore << "\n";
   }
```

While the code runs, data is lost, specifically the value to the right of the decimal point. .83. The fractional part of the number cannot be stored in *testScore,* that variable being a whole number.

Before continuing with the rest of arithmetic operators, let us discuss the concept of constants.

Constants

A constant is a memory location with unchangeable value, any attempt to change constant value will cause an error. A constant is declared with **const** keyword, as follows:

Syntax:

const *datatype constantname=value*;

A constant must be initialized in the declaration statement; otherwise its value will remain rubbish data for the rest of the program, because it is the only place where a constant can be given a value is the definition statement. Following are some examples of constant definitions:

const int x=8;

const double PI=3.1415;

if we tried to change the value of either x or PI an error will occur

PI=3.14156;

This statement will cause an error. a constant value can not be changed, and the assignment operator (=) must have a variable on its left side.

Arithmetic Operators

An operator is a symbol that represents a specific action. We have discussed and used operators discussed earlier, including the assignment operator, =. C++ also supports operators for arithmetic, specifically addition, subtraction, multiplication, and division. Operators used for arithmetic are called, naturally enough, arithmetic operators. Table 2 summarizes them.

Table 2: Arithmetic Operators

Operator	**Purpose**	**Example**	**Result**
+	Addition	5 + 2	7
-	Subtraction	5 – 2	3
*	Multiplication	5 * 2	10
/	Division (Quotient)	5 / 2	2
%	Division (Remainder)	5 % 2	1

The % operator, also called the *modulus* operator, may look unfamiliar. It returns the remainder in division.

Arithmetic operators are binary operators because they operate on two *operands*, binary being a reference to 2, and operand referring to each of the two values that is in the arithmetic expression. For example, in the expression *5 + 2*, the + sign is the operator, and the 5 and 2 each is an operand.

Not all operators are binary. For example, in the expression –3, the negative sign, or negation operator, is a unary operator because it operates on only one operand, which is the integer 3 in this example. There also are ternary operators, which operate on 3 operands. However, all arithmetic operators involve two operands—no more, no less.

The arithmetic operators work with negative as well as positive numbers, and, with the exception of the modulus operator, floating point numbers (numbers with values to the right of the decimal point) as well as whole numbers. The addition operator also works with strings as well as with numbers.

The Addition Operator

At the community college where I teach computer science, students often will pre-register, enrolling in a course before the semester starts. However, some students will add a course during the first few weeks of the semester.

The following program has two integer variables, *total* and *added.* The program first assigns to *total* the value inputted by the user for the number of preregistered students. The program then assigns to *added* the value inputted by the user for the number of students adding the course. Afterward, the program uses the addition operator to add two operands, *total* and *added.* The resulting sum is then assigned to *total*, which now reflects the number of all students in the course, both preregistered and added. That sum then is outputted.

```
#include <iostream.h>
int main(void)
{
   int total, added;
   cout << "Enter number of pre-registered students: ";
   cin >> total;
   cout << "Enter number of students adding the course: ";
   cin >> added;
   total = total + added;
   cout << "Total number of students: " << total;
}
```

The input and output of the program could be

```
Enter number of registered students: 30
Enter number of students adding the course: 3
Total number of students: 33
```

Combined Assignment and Arithmetic Operator

New programmers sometimes are confused by statements such as *total = total + added* because, in *mathematics,* a variable cannot *equal* itself plus another number. However, this statement is not made in mathematics, but in C++ programming, in which the = operator is not used for equality, but instead for assignment.

Nevertheless, there also is another way to express *total = total + added:*

```
   total += added;
```

To the compiler, it makes no difference whether you use *total = total + added* or *total += added.* However, many programmers prefer *total += added,* some because it looks more elegant, others because it seems more readable, and still others for the practical reason that it requires less typing.

This compact form of combining arithmetic and assignment operators is not limited to the addition operator. As Table 3 shows, it also can be used with the other arithmetic operators. In that table, it is assumed *a* is a previously declared integer variable.

Table 3: Combining Arithmetic and Assignment Operators

Statement	Combining Operators
a = a + 2;	a +=2;
a = a – 2;	a –=2;
a = a * 2;	a *=2;
a = a / 2;	a /=2;
a = a % 2;	a %=2;

Precedence between Arithmetic and Assignment Operators

The statement *total = total + added* uses two operators, assignment and arithmetic. The arithmetic operation has *precedence* over the assignment operation. This means the addition is performed before the assignment. This makes more sense intuitively than the other order. Precedence also arises when more than one arithmetic operator is used in a statement.

The Subtraction Operator

At the community college where I teach, students leave the class as well as join it. Some of the preregistered students never show up. Other students who show up later decide to drop the course.

The following program builds on the one we used with the addition operator by adding one integer variable, *dropped* for students who I dropped because they never showed up, or who dropped themselves from the course. The *dropped* variable is assigned a value by the user. The program then uses the subtraction operator to update *total.*

```
#include <iostream.h>
void main(void)
{
   int total, added, dropped;
   cout << "Enter number of pre-registered students: ";
   cin >> total;
   cout << "Enter number of students adding the course: ";
   cin >> added;
   total = total + added;
   cout << "How many students dropped? ";
   cin >> dropped;
   total -= dropped;
   cout << "Total number of students: " << total << endl;
}
```

The input and output of the program could be

```
Enter number of pre-registered students: 30
Enter number of students adding the course: 3
How many students dropped? 5
```

```
Total number of students: 28
```

In this example, we used the combined assignment and arithmetic operator –= in the expression *total –= dropped*, rather than *total = total – dropped*. As explained with the addition operator, either alternative will work the same way.

The effect of overflow and underflow are the same with the subtraction operator as with the addition operator. However, unlike the addition operator, the subtraction operator will not work with string operands.

The Multiplication Operator

Returning to my community college course example, all students who enroll in a course owe a tuition of $72, even if they don't show up or later drop the course.

The following program builds on the one we used with the addition operator by adding the following statement:

```
cout << "Total tuition owed: $" << (total + dropped) * 72
     << endl;
```

The program now reads

```
#include <iostream.h>
void main(void)
{
   int total, added, dropped;
   cout << "Enter number of pre-registered students: ";
   cin >> total;
   cout << "Enter number of students adding the course: ";
   cin >> added;
   total = total + added;
   cout << "How many students dropped? ";
   cin >> dropped;
   total -= dropped;
   cout << "Total number of students: " << total << endl;
   cout << "Total tuition owed:
  $" << (total + dropped) * 72
        << endl;

}
```

The input and output of the program could be

```
Enter number of preregistered students: 30
Enter number of students adding the course: 3
How many students dropped? 5
Total number of students: 28
Total tuition owed: $2376
```

The variables *total* and *dropped* are added so that *total* reflects all students ever enrolled, even if they are no longer in the class, because all students owe tuition even if they don't show up or later drop the course.

The effect of overflow and underflow are the same with the multiplication operator as with the addition and subtraction operators. Unlike the addition operator, but like the subtraction operator, the multiplication operator will not work with string operands.

Precedence Between Arithmetic Operators

The statement we added has two arithmetic operators, for addition and multiplication. The order in which the two arithmetic operations are performed makes a difference. If addition is performed first, 28 + 5, and then the sum, 33, is multiplied by 72, the result is 2376. However, if multiplication is performed first, 5 * 72, and then the product, 360, is added to 28, the result would be 388.

C++ has rules, called *precedence,* for determining which operation is performed first. Precedence was discussed earlier. However, following table lists the precedence between arithmetic operators.

Table 4:Precedence of Arithmetic Operators

Precedence	**Operator**
Highest	– (unary negation)
Middle	* / %
Lowest	+ –

When there is more than one operator in a row, and those operators have equal precedence. Thus, the multiplication operator and the two division operators have equal precedence. Similarly, the addition and subtraction operators have equal precedence.

Table 5 shows the results of applying precedence to several arithmetic expressions. We have not reviewed the division operators yet, but in the examples in Table 5 the / operator works exactly as it does in arithmetic.

Table 5: Precedence in Action

Expression	**Result**
2 + 3 * 4	14, not 20
8 / 2 – 1	3, not 8

C++ also has rules called *associativity* for determining which operation is performed first when two operators have equal precedence. Table 6 describes those rules.

Table 6: Associativity of Arithmetic Operators

Operator	Associativity
(unary negation)	Right to left
* / %	Left to right
+ –	Left to right

Therefore, the result of the expression 8 / 2 * 4 is 16, not 1, because division, being the leftmost operator, is performed first.

However, there are times when you want to override the default precedence. For example, in our program, in calculating tuition, the default precedence would be to multiply *dropped* by 72, after which the product would be added to *total.* However, we want to change the order of operations so that *dropped* is first added to *total,* and the sum then multiplied by 72.

You can override the default precedence with parentheses. This is done in the statement:

```
cout << "Total tuition owed: $" << (total + dropped) * 72
     << endl;
```

Expressions in parentheses are done first. As a result of the parentheses, the expression *(total + dropped) * 72* is evaluated so that dropped is first added to *total,* and the sum is then multiplied by 72.

Division Operators

Addition, subtraction, and multiplication each have one operator. However, division has two. The / operator gives you the quotient, while the % (or modulus operator) gives you the remainder.

Quotient and remainder, along with dividend and divisor, are terms that I first learned in elementary school and then did not use very much again until many years later. If you are rusty on your arithmetic terminology like I was, this example may help. In the problem 7 divided by 2, 7 is the dividend and 2 is the divisor. The result of this division is that 3 is the quotient and 1 is the remainder.

The Division Operator

The division operator returns the quotient. However, the value of the quotient depends on whether at least one of the operands is a floating point data type.

For example, the value of 10 / 4 is 2.5. However, in C++, the value is 2 because, when both operands are an integer or other whole number data type, then the result is an integer as well, and the remainder is not part of the quotient. This is true even if the result is assigned to a floating point variable. The output of the following program is 10 / 4 = 2.

```
#include <iostream.h>
void main(void)
{
   int firstOp = 10, secondOp = 4;
   float result = firstOp / secondOp;
   cout << firstOp << " / " << secondOp << " = " << result;
}
```

However, the value of 10.0 / 4 is 2.5 in C++. When at least one of the operands is a floating point data type, and 10.0 would be interpreted as floating point, then the result is a floating point as well. The output of the following program is 10 / 4 = 2.5 because we changed the data type of *firstOp* from int to float:

```
#include <iostream.h>
void main(void)
{
   float firstOp = 10, result;
   int secondOp = 4;
   result = firstOp / secondOp;
   cout << firstOp << " / " << secondOp << " = " << result;
}
```

Going back to the first example, if you want the result of the division of two integer variables to be a float, then you have to *cast* one of the variables to a float. A cast does not change the data type of the variable, just the data type of the *value* of the variable during the completion of the operation. You cast the variable by putting the desired data type in front of it in an expression, and placing either the desired data type or the variable in parentheses. This is how the first example could be changed to make the result of integer division a float:

```
#include <iostream.h>
void main(void)
{
   int firstOp = 10, secondOp = 4;
   float result = (float) firstOp / secondOp;
   cout << firstOp << " / " << secondOp << " = " << result;
}
```

All of the following expressions would work

```
float result = (float) firstOp / secondOp;
float result = float (firstOp) / secondOp;
float result = firstOp / (float) secondOp;
float result = firstOp / float (secondOp);
```

However, in some programs you may want integer division so that the quotient will ignore the fractional value.

.

The Modulus Operator

The modulus operator also involves division, but returns only the remainder. For example, the result of *7 % 2* is not the quotient, 3, but the remainder, 1.

The modulus operator works only with whole number operands. The result of an attempt to use it with a floating point operand is undefined. The result often is a compiler error, but this is compiler dependent.

When you use the / or the % operator, you cannot divide by zero. The result is an error.

Increment and Decrement Operators

Increment means to increase a value by one. Conversely, decrement means to decrease a value by one. C++ has an increment operator that you can use to increase a value by one and a decrement operator that you can use to decrease a value by one. This section will show you how to use both, something that will be useful in the next section on the for loop, which uses increment and decrement operators.

The Increment Operator

In the following program, the statement *num += 1* increases the value of the integer variable *num*, which was initialized to the value 2, by 1, so the output will be 3.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   num += 1;
   cout << num;
   return 0;
}
```

Another way to accomplish the same result is by using the increment operator, ++. The increment operator is unary—that is, it operates on one operand. That operand generally is a whole number variable, such as an int. We can use the increment operator simply by changing the program we just ran by replacing the statement *num* += 1 with the statement *num*++:

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   num++;
   cout << num;
```

```
   return 0;
}
```

The same output would occur if you substituted the statement ++*num* for *num*++:

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   ++num;
   cout << num;
   return 0;
}
```

Placing the ++ before the variable *num* is called prefix incrementing—the "pre" indicating that the increment operator precedes its operand. Placing the ++ before the variable *num* is called postfix incrementing—the "post" indicating that the increment operator follows its operand.

In this example, it makes no difference to the output of the program whether you use prefix or postfix incrementing. The reason is that the statement ++*num* has only one operator; the same is true of the statement *num*++. However, there is a difference between prefix and postfix incrementing when the statement has more than one operator. This is discussed later in this chapter in the section "The Difference Between Prefix and Postfix."

The Decrement Operator

In the following program, the statement *num* –= *1* decreases the value of the integer variable *num*, which was initialized to the value 2, by 1, so the output will be 1.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   num -= 1;
   cout << num;
   return 0;
}
```

Another way to accomplish the same result is by using the decrement operator, --. The decrement operator, like the increment operator, is unary, operating on one operand which generally is a whole number variable, such as an int. We can use the decrement operator simply by changing the program we just ran and replacing the statement *num* –= *1* with the statement *num*--:

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

```
int main(void)
{
   int num = 2;
   num--;
   cout << num;
   return 0;
}
```

The same output would occur if you substituted the statement --*num* for *num*--:

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   --num;
   cout << num;
   return 0;
}
```

As with the increment operator, placing the -- before the variable *num* is called prefix decrementing, while placing the -- after the variable *num* is called postfix decrementing.

Also, as with the example of the increment operator, in this example it makes no difference to the output of the program whether you use prefix or postfix decrementing because the statement --*num* (or *num*--) has only one operator. However, as discussed in the next section, "The Difference Between Prefix and Postfix," there is a difference between prefix and postfix decrementing (or incrementing) when the statement has more than one operator.

The Difference Between Prefix and Postfix

The following program is similar to the previous program that illustrated the increment operator.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   cout << num++;
   return 0;
}
```

However, instead of two statements:

```
   num++;
   cout << num;
```

This program uses one statement:

```
   cout << num++;
```

There are two operators in this cout statement: the increment operator ++ and the stream insertion operator <<. The issue is one of precedence; which operation occurs first.

The output of this program is 2. The reason is that when an increment or decrement operator is postfix, that operation is the *last* to occur. Therefore, the output of *num* occurs first while the variable's value is still 2, and then the value of *num* is incremented from 2 to 3.

Now, change the line:

```
   cout << num++;
```

to the line:

```
   cout << ++num;
```

so the program now reads

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 2;
   cout << ++num;
   return 0;
}
```

This time, the output of this program is 3 instead of 2. The reason is that when an increment or decrement operator is prefix, that operation is the *first* to occur. Therefore, *num* first is incremented from 2 to 3 before the value of *num* is outputted.

The distinction between prefix and postfix also arises frequently with arithmetic operators. In the following code fragment, the value of *result* is 15, not 18, because *op*2 is incremented from 5 to 6 after the multiplication and assignment occurs.

```
int op1 = 3, op2 = 5, result;
result = op1 * op2++;
```

If prefix incrementing instead were used, as in the following code fragment, the value of *result* is 18, not 15, because *op2* is incremented from 5 to 6 before the multiplication and assignment occurs.

```
int op1 = 3, op2 = 5, result;
result = op1 * ++op2;
```

The distinction between prefix and postfix arises as well with relational operators. In the following code fragment, the output is 1 (the integer representation of Boolean true)

because the integer variable *num* is compared to 5 for equality before *num* is incremented from 5 to 6.

```
int num = 5;
cout << (num++ == 5);
```

If prefix incrementing instead were used, as in the following code fragment, then the output would be 0 (the integer representation of Boolean false) because the integer variable *num* is incremented from 5 to 6 before it is compared to 5 for equality.

```
int num = 5;
cout << (++num == 5);
```

The increment and decrement operators generally are not used by themselves, but in conjunction with loops. The next section covers one type of loop: the for loop.

Operators' Complete listing:

You saw, the use of the + sign. This is an *operator*. An *operator* is simply some symbol that performs some action, usually a mathematical action, such as addition or subtraction. C++ supports a number of important operators that you will need to get familiar with. Let's begin examining C++ operators, starting with the basic math operators, shown in Table 7.

Table 7: Operators

Operator	**Purpose**	**Example**
+	To add two numbers	int answer; answer = 5 +6;
-	To subtract two numbers	int answer; answer = 10 –3;
*	To multiply two numbers	int answer; answer = 4 * 5;
/	To divide two numbers	int answer; answer = 7/3;
++	This is a special operator that simply ments the value by 1. You will see this used later in this book when loops are discussed.	int answer; incre answer++;
--	This is also a special operator that simply decrements the value by 1.	int answer; answer--;
=	The single equals sign is an assignment operator. It states "make what's on the left equal to what's on the right"	answer = 16;
==	The double equals is an equality	if(answer==5)

Table 7: Operators

Operator	Purpose	Example
	operator. It asks “is what’s on the left equal to what’s on the right?” This is frequently used in if statements	
!=	Not equal to	if (x !=3)
+=	Add then assign	x += 1;
-=	Subtract then assign	x -=1;
||	This is the logical OR operator.	if(j == 5 || j ==10)
&&	This is the logical AND operator	if(j > 5 && j<10)
>>	Bitwise shift to the right	3<<2;
<<	Bitwise shift to the left	3>>2;
&	Bitwise And	3&2;
|	Bitwise Or	3|2

note `When using the increment (++) or decrement (—) operator, it is necessary to be careful of where you put it. If you put the operator before the variable, then that operation will take place before any other operations in the expression.`

math.h functions

C++, unlike some other programming languages, does not have an exponent operator. Instead, it has a built-in function named *pow*, which is defined in the standard library *cmath.* The name *pow* is shorthand for power, since with exponents one number is raised to the power of another.

The *pow* function has two arguments. The first argument is the number that is being raised to a certain power. The second argument is the power the first argument is being raised to. Therefore, the expression *pow (4, 2)* would be used to raise 4 to the power of 2, the result being 16.

While in the example 4 to the power of 2, the result is a whole number, the *pow* function returns a double data type. Floating point numbers also can be raised to a power, resulting in another floating point number. Additionally, whole numbers can be raised to a negative power, which also may result in a floating point number.

The *pow* function is useful for solving math problems. The formula for the area of a circle is $area = \pi r^2$. Assuming a value of π of 3.14159, two double variables *area* and *radius,* and that radius has already been assigned a value, the code for determining the circle’s area is

```
area = 3.14159 * pow(radius, 2);
```

The following program calculates the area of a circle based on a radius inputted by the user.

```
#include <iostream.h>
#include <math.h>
void main(void)
{
   double radius, area;
   const double PI=3.14159;
   cout << "Enter radius of circle: ";
   cin >> radius;
   area = PI* pow(radius, 2);
   cout << "The area is " << area << endl;

}
```

The input and output could be

```
Enter radius of circle: 6
The area is 113.097
```

Other functions can be found in

Math Functions

Function	**Purpose**
double cos(double);	This function takes an angle (as a double) and returns the cosine.
double sin(double);	This function takes an angle (as a double) and returns the sine.
double tan(double);	This function takes an angle (as a double) and returns the tangent.
double log(double);	This function takes a number and returns the natural log of that number.
Double pow(double, double);	With this function, you pass it two numbers. The first is a number you wish to raise and the second is the power you wish to raise it to.
double hypot(double, double);	If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse.
double sqrt(double);	You pass this function a number and it gives you this square root.
int abs(int);	This function returns the absolute value of an integer that is passed to it.
double fabs(double)	This function returns the absolute value of any decimal number passed to it.

Math Functions	
Function	**Purpose**
double floor(double)	Finds the integer which is less than or equal to the argument passed to it.

Control Structures in C++

Index

Control Structures in C++

3.1 Relational Operators

A relational operator is used to make comparison between two expressions. All these operators are binary and require two operands.

Operators	Meaning	Usage
= =	Equal to	Expr1 = = expr2
! =	Not equal to	Expr1 ! = expr2
<	Less than	Expr1 < expr2
< =	Less than or equal to	Expr1 < = expr2
>	Greater than	Expr1 > expr2
>=	Greater than or equal to	Expr > = expr2

Each one of the operators *compares* its left hand side (**LHS**) operand with the right hand side (**RHS**) operand. The whole expression involving the relational operator then evaluates to an integer. It evaluates to zero if the condition is *false*, and non-zero value, if it is *true*.

Example

Program to demonstrate usage of relational operators.

```
#include <iostream>
using namespace std;
int main(void)
{
   int a = 4, b = 5;
   cout << a << " > " <<  b << " is " << (a > b) << endl;
   cout << a << " >= " <<  b << " is " << (a >= b) << endl;
   cout << a << " == " <<  b << " is " << (a == b) << endl;
   cout << a << " <= " <<  b << " is " << (a <= b) << endl;
   cout << a << " < " <<  b << " is " << (a < b) << endl;
   return 0;
}
```

Output

The program's output is

```
4 > 5 is 0
4 >= 5 is 0
4 == 5 is 0
4 <= 5 is 1
4 < 5 is 1
```

3.2 Logical Operators

An expression that evaluates to zero denotes a *false* logical condition and that evaluating to non-zero value denotes a *true* logical condition. An expression that combines one or more relational operators is called a *logical expression*. Logical operators are useful in combining one or more conditions.

C++ supports three logical operators shown below.

Operators	Meaning	Usage	example
&&	Logical AND	Expr1 && Expr2	a && b
||	Logical OR	Expr1 || Expr2	a || b
!	Logical NOT	!Expr	!a

Operator **&&** and || are binary operators and **!** is a unary operator.

The table below is the **truth table** for logical operators.

Expr1	Expr2	Expr1 && Expr2	Expr||Expr2	!Expr1
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Example

Expression	Logical Value
!(20==12+8)	False
(4>=5)&&(10==(6+4))	False
(34>=5)&&(10==(6+4))	True
(3>7) || (34<42)	True

Example
Evaluate the following logical expression
!((-5.0 >= -6.2) || ((7 != 3) && (6 == (3 + 3))))
!(true || (true) && (6 == 6))
!(true || (true) && (true))
!(true || true)
! true
False

Quiz 1:
1. Assume that x= 12.5 and Flag = true
A= (x != 7/3) || !((x>= 4) && (!flag))
   a. A=True
   b. A=False

2. Assume That x= 5, y=2 and z=10
B= x + y < 10 && x/y == 3 || z != 10
   a. B=True
   b. B=False

Note: Confusing equality (==) and assignment (=) operators

Ex:
1. X=4, put 4 in X.
2. X==4, is X equal 4.

Control Structures in C++

3.3 Conditional Operator

The conditional operator ***?:*** takes 3 operands.

It has the following syntax.

```
expr1 ? expr2 : expr3;
```

expr1, expr2 & expr3 may be any valid variable, constant or an expression.

Working of conditional operator

Example

Program to demonstrate usage of conditional operator.

```
#include < iostream.h>

int main()

{

int num;

cout << " enter the number : ";

cin >> num;

(num % 2) ? cout << " Odd\n " : cout " Even\n ";

}
```

Output

```
Enter the number : 24

Even.

Enter the number : 25

Odd.
```

3.4 Control Structures

Statements in a program are usually executed one after the other in the order in which they are written. But sometimes, the next statement to be executed may be other than the next one in sequence. This is called **transfer of control**.

The transfer of control could be based on a condition being met. Sometimes the statements are executed in a loop, i.e., the same set of statements is repeated again and again.

All the programs could be written in terms of three control structures, namely

- **Sequence structure**
- **Selection structure** and
- **Repetition structure**

Sequence Structure is built into C++, – unless directed otherwise,the computer executes C++ statements one after the other in the order in which they are written.

Sequence, Selection and Repetition Constructs

Let us take an example for the sequential flow:

English = 60
Maths = 7
TotalMarks = English + Maths
Display “Total Marks = “ + TotalMarks

The code snippet for the above would be as follows:

```
#include < iostream.h >

# include < conio.h >

main ()

{

  int EngMarks=0;

  int MathMarks=0;

  int TotMarks=0;

  cin>>EngMarks;

  cin>>MathMarks;

  TotMarks=EngMarks+MathMarks;

   cout <<"Total Marks="<<TotMarks;

  return(0);

}
```

The **output** in this example would be the total marks secured in English and Maths combined

Selection Structure:

A selection structure is used to choose among alternative courses of action. C++ provides three types of selection structures. They are,

Repetition Structure

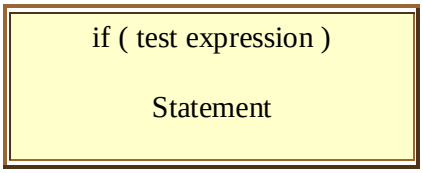
A body of statements can be repeatedly executed, either conditionally or for a certain number of times.

Loops are used for operations that are to be repeated some number of times. The loop repeats until some specified condition at the beginning or end of the loop is met.

A repetition structure allows the programmer to specify that an action is to be repeated based on the truth or falsity of some condition. If the action is true then the action is performed. This action will be performed repeatedly while the condition remains **true**. When the condition is **false**, the loop is terminated.

3.5 If Structure

It is a powerful decision making structure and is used to control the flow of execution of statements. It is basically a two-way decision statement and in conjunction with an expression.



An *if statement* provides for the conditional execution of a statement block based on whether a specified expression is ***true***.

If structure determines whether particular statement(s) have to be performed or not. If the condition is **True**, one statement or a group of statements is performed and moves on to the next program statement in order. If the condition is **False**, then it skips the execution of the statement(s) mentioned above and moves on to the next program statement in order.

The **flowchart** for the **If** structure is given below.

The flow chart illustrates the single selection *If structure*. The flowchart contains a diamond symbol, also called decision symbol, which indicates that a decision is to be made. The decision symbol contains an expression, such as a condition that is either **True** or **False**.

Thus, **If** selection structure performs an indicated action only when the condition is **True**; otherwise the action is skipped.

For example, suppose the passing grade in an exam is 60. The statement.

If student's grade is greater than or equal to 60.

Display "Passed"

determines if the condition "student's grade is greater than or equal to 60" is **True** or **False**.

If the condition is **True** then "Passed" is displayed and the next program statement in order is performed.

If the condition is **False**, the display statement is ignored and the next program statement in order is performed.

The C++ code snippet can be written as,

```
…
if (Grade >= 60)
cout << "Passed";
…
```

The **output** here would be "Passed" if Grade is 60 or more.

If structure is also called **Single Selection Structure**.

In some instances, it's necessary to perform several statements when condition is true, in this case, several simple statements can be written as a single compound statement (Block).
The general form of block single selection if statement is:

If (Condition)

{

Statement1;

Statement2;

. . .

Statement N;

}

Example 1:

```
If (grade>60)
{Cout<< "Your Marks is";
  Cout<< grade<<endl;
  Cout<<"Pass";
}
```

The **output** here would be
Your Mark is grade
Pass

If Grade is 60 or more

Example 2:

```
If (a<0 && b>0)
{ Cout<< " Your first number is "<<a;
   Cout<<" Negative\n";
   Cout<<"Your Second number is "<<b;
   Cout<<" Positive"<<endl;
}
```

The **output** here would be
Your first number is (value of a) Negative
Your Second number is (value of b) Positive

If a less than zero, and b greater than zero.
Example 3:

```
If (a<0)
{ Cout<< "your first number is"<<a;
   Cout<<"Negative\n";
   If (b > 0)
         Cout<<"Your Second number is"<<b;
         Cout<<"Positive"<<endl;
}
```

The **output** here would be
If a is less than zero.
Your first number is (value of a) Negative
And b is greater than zero.
Your Second number is (value of b) Positive

3.6 If else Selection Structure

The syntax for if statement is.

```
if ( condition )
statement1
else
statement2
```

The condition must be enclosed in parentheses.

The **If … else** structure allows the programmer to specify that a different action is performed when the condition is **True** than when the condition is **False**.

Here, if the condition is **True**, then statement(s) 1 is performed and moves on to the next program statement in order. If the statement is **False**, then statement(s) 2 is performed and then moves on to the next program statement in order.

Example:

If student's grade is greater than or equal to 60

Display "Passed"

Else

Display "Failed.

The above statement displays "Passed" if the student's grade is greater than equal to 60 and displays "Failed" if the student's grade is less than 60.In either case, after display occurs, the next program statement in sequence is performed.

The flow chart for the same is given below.

The C++ code snippet would be

```
...
If grade >= 60
cout << "Passed";
```

```
else
Cout << "Failed";
…
```

Output:

The output here would be “Passed” if grade is 60 or more and “Failed” if grade is less than 60.

If… else structure is also called **Double Selection Structure**.

Quiz:

1. Final value of sum is =?, if X= 2 and Sum=0.
If (x>0)
Sum +=x;

a. **0**
b. **2**

2. Count =0, X=-2 and Sum=0, what is the final value of Count =?.
If (X>0.0)
Sum += X;
Count++;

a. **0.**
b. **1**

3. Final value of Num =?
Cin>>num;
If (num<0)
Num+=2;
Num ++;
Cou<< Num;
If Num input = 2

a. **4**
b. **3**

4.

```
If (Num3>=2)
{   Cout<< Num3;
    Total = Total + Num3;
    Num3 = Num3 – 2;
    Cout<<Num3<<endl;
}
Cout<<Total;
```

If Num3=5, Output is

a. **535**

b. 53
5

Blocked if/else structures

A block (or compound statement) can be used in the if/else structure. The general form:

```
If (Condition)
{
      Statement1;
       . . .
       Statement N;
}
Else
{
        Statement1;
         . . .
        Statement N;
}
```

Example

```
If (grade>60)
{Cout<< "Your Marks is";
  Cout<< grade<<endl;
  Cout<<"Pass";
}
Else
{Cout<< "Your Marks is";
  Cout<< grade<<endl;
  Cout<<"Fail";
}
```

The output would be as

Your Mark is (value of grade)
Pass
If grade greater than 60
Otherwise
Your Mark is (value of grade)
Fail

Quiz:

1. X has the value 3.5 when the following code executed what value would be assigned to y?

```
If (x + 1 <= 3.6)
    Y= 1.0;
Else
    Y= 2.0;
```

a. 1.0
b. 2.0

2. X = 10, Y = 5 and sum = 0

```
If (X >= Y)
    Sum += X;
    Cout<< "X is bigger" << endl;
Else
    Sum += Y;
    Cout<< "Y is bigger" << endl;
```

After the previous code is executed the output is

a. X is bigger
b. Y is bigger
c. Error

3. Assume X = 4, Y= 6

```
If (x < 5 )
If ( y > 5 )
    Cout<<  "Jordan"<<endl;
Else
    Cout << "Amman"<<endl;
    Cout << "Jerash"<<endl;
```

Outpur is:

a. Jordan
   Amman
b. Jordan
   Jerash
c. Amman
   Jerash

4. Assume X = 6, Y= 4

```
If (x < 5 ) {
```

```
    If ( y > 5 )
        Cout<<  "Jordan"<<endl;}
    Else {
        Cout << "Amman"<<endl;
        Cout << "Jerash"<<endl;}
```

Outpur is:

d. **Jordan**
   Amman
e. **Jordan**
   Jerash
f. **Amman**
   Jerash

3.7 Multiple If or Nested If Selection Structure

Nested If structures test for multiple cases by placing one **If… else** structure inside another **If … else** structure. The general form is as follows.

```
If (Condition1)
        Statement1;
Else if ( Condition2 )
        Statement2;
. . .

Else if ( Condition-n )
        Statement-n;
Else
        Statement-e;
```

Example 1:

```
If( Condition1==True)
        If (condition2=True)
                Cout<<"  Condition1 true, Condition2 true";
        Else
                Cout<<"Condition1 true, Condition2 false";
Else
        Cout<<"Condition1 False";
```

Example 2:

```
If( Condition1==True)
{       If (condition2=True)
                Cout<<"  Condition1 true, Condition2 true";
        Else
                Cout<<"Condition1 true, Condition2 false";
}
Else
```

Cout<<"Condition1 False";

Example 3:

```
If (x < 0)
        Sign = -1;
If (x == 0)
        Sign = 0;
If (x > 0)
        Sign = 1;
```

Example 4:

Print **A** for exam grades greater than or equal to 90, **B** for grades in the range 80 to 89, **C** for grades in the range 70 to 79, **D** for grades in the range 60 to 69, **F** for all other grades.

This program can be written in C++ as follows:

```
…
if (grade >= 90)

cout << "A";

else if (grade >= 80)

cout << "B";

else if (grade >= 70)

cout << "C";

else if (grade >= 60)

cout << "D";

else

cout << "F";
```

If the grade is greater than or equal to 90, the first four conditions will be True, but always the statement after the first test will be executed. After the assignment is executed, the else part of the 'outer' if… else statement is skipped, in fact skipping the entire reminder of the preceding code segment.

Output

Depending on the grade secured, the grade is display.

Quiz:

1.

```
If (L=='a' || L=='A')
        Cout<<"Vowel Letter";
If (L=='e' || L=='E')
        Cout<<"Vowel Letter";
Else if (L=='i' || L=='I')
else
        Cout<<"Consonant Letter";
```

Assume L='b' output is

a. **Consonant Letter**
b. **Vowel Letter**
c. **Nothing**

2.

```
If (L=='a' || L=='A'){
        Cout<<"Vowel Letter"
If (L=='e' || L=='E')
        Cout<<"Vowel Letter";}
Else if (L=='i' || L=='I')
        Cout<<"Vowel Letter";
else
        Cout<<"Consonant Letter";
```

Assume L='E' output is

A. **Consonant Letter**
B. **Vowel Letter**
C. **Nothing**

3.

```
If (L=='a' || L=='A')
{       Cout<<"Vowel Letter"
If (L=='e' || L=='E')
        Cout<<"Vowel Letter";
Else if (L=='i' || L=='I')
```

} Cout<<"Vowel Letter";
else
Cout<<"Consonant Letter";
Assume L='a' output is

A. Consonant Letter
B. Vowel Letter
C. Error

3.8 Switch Case Construct

We have discussed the **If** single selection structure and the **If… else** double selection structure. We have also discussed multiple selection using the **nested If** structure. C++ provides an alternative method for choosing among a set of mutually exclusive choices through *switch…case… construct*.

Although **If… else…** multiple case structure and **Switch… Case** multiple selection structures perform equivalently, many programmers prefer to use **Switch… Case**. It can be used with any data type and its flowchart is given below.

A switch statement consists of the following components.

```
switch ( expression )
{
case constant1: stat1;
break;
case constant2: stat2;
break;

……..
……..
default:    stat n;
break;
}
```

- A switch keyword followed by an expression in parentheses that is to be evaluated.

- A set of case labels consisting of keyword case followed by a constant expression against which to compare the result of the switch expression, followed by a colon.
- A sequence of program statements is with associated with set of case labels.
- An optional default label is used. If switch does not match with any of the case labels, the statements following the default label are executed.

Example 1:

```
Switch (day) {
Case 1: Cout<<"Sunday"<<endl;
        Break;
Case 2: Cout<<"Monday"<<endl;
        Break;
Case 3: Cout<<"Tuesday"<<endl;
        Break;
Case 4: Cout<<"Wenesday"<<endl;
        Break;
Case 5: Cout<<"Thursday"<<endl;
        Break;
Case 6: Cout<<"Friday"<<endl;
        Break;
Case 7: Cout<<"Saturday"<<endl;
        Break;
default: Cout<<"invalid day"<<endl;
        Break;
```

If you removed the break statements from the program, you could have the following sample run:

```
Sunday
Monday
Tuesday
Wenesday
Thursday
Friday
Saturday
invalid day
```

Example 2:

```
Switch (day) {
Case 1:
Case 7: Cout<<"This is a weekend day";
```

```
        Break;
Case 2:
Case 3:
Case 4:
Case 5:
Case 6: Cout<<"This is a weekday";
        Break;
default: Cout<<"invalid day";
        Break;
}
```

Example 3:

An example written in C++ using **Switch… Case** structure is shown below.

```
#include < iostream.h >
main ( )
{
char ch;
cin >> ch;
switch ( ch )
        {
          case ’a’:
          case ’A’:
          case ’e’:
          case ’E’:
          case ’i’:
          case ’I’:
          case ’o’:
          case ’O’:
          case ‘u’:
          case ‘U’: cout << “This is a vowel.”;
                          break;
          default: if ( isalpha ( ch ) )
                        cout << “This is a consonant.”;
                          break;
           }
}
```

Output

If the user enters a vowel, the output is
“This is a vowel.”
and if the user enters a consonant, the output is
“This is a consonant.”

Quiz:

1.
```
Switch (value % 2){
Case 0:
        Cout<<"Even integer"<<endl;
Case 1:
        Cout<<"Odd integer"<<endl;}
```

Assume value = 4 output is
a. Even integer
b. Odd integer
c. Even integer
   Odd integer

2.
```
Switch (value % 2)
Case 0:
        Cout<<"Even integer"<<endl;
Case 1:
        Cout<<"Odd integer"<<endl;}
```

Assume value = 4 output is
a. Even integer
b. Even integer
   Odd integer
c. Error

3.9 Looping Constructs

Every computer language must have features that instruct a computer to perform repetitive tasks. The process of repetitive execution of a block of statements is known as **looping**.

In looping, a sequence of statements is executed until some conditions for the termination of the loop are satisfied. A program loop therefore consists of two segments, one knows as *body of the loop* and the other known as the *control statement*. The control tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled* loop, or as the *exit-controlled loop*.

- In the **entry-controlled loop**, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed.

- In the case of **exit-controlled loop**, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

3.10 For loop Structure

If you wanted to output the numbers between 1 and 10, you could write a program such as the following:

```
 #include <iostream>
using namespace std;
int main(void)
{
   int num = 1;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   cout << num++;
   return 0;
}
```

However, you could write the same program with far less code by using a for loop:

```
#include <iostream.h>
int main(void)
{
   for (int num = 1; num <= 10; num++)
      cout << num << " ";
   return 0;
}
```

The difference between the two programs becomes more pronounced if you change the specification from outputting the numbers between 1 and 10 to outputting the numbers between 1 and 100. I won't rewrite the first program because it would take up too many pages; suffice it to say, you would have to add 90 more cout statements. However, the same program using a for loop would be:

```
#include <iostream.h>
int main(void)
{
   for (int num = 1; num <= 100; num++)
      cout << num << " ";
   return 0;
}
```

Indeed, by using the for loop, the same code could output the numbers between 1 and 1000 or even 1 and 10000; you just would need to change the 100 in the code to 1000 or 10000.

The For structure handles all the details of counter-controlled repetition. Let us consider the example of counter to count even numbers from 2 to 20. The program logic using For structure is written as follows.

```
For counter = 2 to 20 step 2
Print counter
Next counter
```

The for loop begin with test of the true condition i.e either loop can terminate before the execution of the associated statement or statement block.

A for loop is used most commonly to step through a fixed – length data structure, such as an array or a vector.

A looping process would include the following four steps.

- Setting and initialization of the counter
- Execution of the statements in the loop
- Test of a specified condition for execution of the loop
- Incrementing the counter

```
The syntax form of for loop is
for ( init-statement; condition;
expression )
```

- *Init-statement* can be either a declaration statement or an expression. It is used to initialize or assign a starting value that is incremented or decremented over the course of the loop
- *Condition servers* as the loop control. As for as condition is true, statement is executed again and again. If the condition is false, statement is never executed
- Expression is *evaluated* after each iteration of the loop

code snippet for the above in C++ would be:

```
#include <iostream.h>
main()
{
    int i;
    . . .
        for (i=2; i<=20; i+=2)
                {
                    cout << i;
                    cout<< endl;
                }
               return (0);
}
```

The output would be:

```
2
4
6
8
10
12
14
16
18
20
```

Some other examples:

- Vary the control variable from 1 to 100 in increments of 1.
  for (i=1; i<=100; i++)
- Vary the control variable from 100 to 1 in increments of –1 (decrements of 1).
  for(b=100; b>=1; b--)
- Vary the control variable from 7 to 77 in increments of 7.
  for(c = 7; c<=77; c+=7)
- Vary the control variable from 20 to 2 in decrements of 2.
  for(d =20; d>= 2; d - =2)
- Vary the control variable over the following sequence of values:
  2,5,8,11,14,17,20.
  for (i=2; i<=20; i+=3)
- Vary the control variable over the following sequence of values:
  99,88,77,66,,44,33,22,11,0
  for(f = 99; f>=0; f– =11)
- Vary the control variable from 1 to 100 in increments of 1.

int num = 1;

for (; num <= 10;)

num++;

- Beware the infinite loop(This loop that never stops executing).

```
#include <iostream.h>
int main(void)
{
   int num = 1;
   for (; num <= 10;)
   {
      cout << num << " ";
   }
   return 0;
}
```

The reason is that the condition *num <= 10* would never become false since *num* would start at 0 and its value would never change because the statement *num++* was omitted.

Nesting For Loops

You can nest a for loop just as you can nest if statements. For example, the following program prints five rows of ten X characters:

```
#include <iostream>
using namespace std;
int main(void)
{
   for (int x = 1; x <= 5; x++)
   {
      for (int y = 1; y <= 10; y++)
         cout << "X";
      cout << '\n';
   }
   return 0;
}
```

The for loop *for (int x = 1; x <= 5; x++)* is the outer for loop. The for loop *for* (*int y = 1; y <= 10; y++*) is the inner for loop.

With nested for loops, for each iteration of the outer for loop, the inner for loop goes through all its iterations. By analogy, in a clock, minutes are the outer loop, seconds the inner loop. In an hour, there are 60 iterations of minutes, but for each iteration of a minute, there are 60 iterations of seconds.

Quiz

1. for (X=100, X>=1, X++)

Cout<< X << endl;

Output will be as follows

a. 1 … 100
b. 1

…

100

c. Error

2. for (X = 19 ; X >= 1 ; X += 2)

Cout<< X <endl;

Output will be as follows

a. 1

…

19

b. 1 . . . 19
c. 1 3 5 7 … 19

3. 1. for (X=100; X>=1; X++)

Cout<< X << endl;

Output will be as follows

a. 1 … 100
b. 1

…

100

c. Infinite loop

3.11 While loop Structure

The while loop is similar to a for loop in that both have the typical characteristics of a loop: the code inside each continues to iterate until a condition becomes false. The difference between them is in the parentheses following the for and while keywords.

The parentheses following the for keyword consists of three expressions, initialization, condition, and update. By contrast, the parentheses following the while keyword consists only of the condition; you have to take care of any initialization and update elsewhere in the code.

This difference is illustrated by the following program that outputs the numbers between 1 and 10.

```
#include <iostream.h>
int main(void)
{
   for (int num = 1; num <= 10; num++)
      cout << num << " ";
   return 0;
}
```

The same program using the while loop could be

```
#include <iostream.h>
int main(void)
{
   int num = 1;
   while (num <= 10)
   {
      cout << num << " ";
      num++;
```

```
    }
    return 0;
}
```

It is a *top-tested or entry-controlled* loop structure. The condition will be tested at the beginning of the loop and the body of the loop is performed only when the condition is met. For as many iterations as the condition evaluates as true, the statement block is executed. If the condition is not satisfied then the control is transferred to the statement after the loop, without executing the body of the loop even once.

The synctactic form of ***while*** loop is

```
while ( condition )
{
statements;
}
```

The sequence is as follows.

1. Evaluate the condition
2. Execute the statement/s if the condition is true
3. Go back to 1

If the first evaluation of the condition yields false, statement is never evaluated.

The Flow chart is given below.

Algorithm:

```
counter = 2
While counter <= 20
Print counter
counter = counter + 2
```

Loop

C++ code snippet

```
. . .
int counter;
. . .
      counter = 2;
           while (counter <= 20)
               {
                 cout << counter;
                 counter = counter + 2;
               }
. . .
```

The output would be:

```
2
4
6
8
10
12
14
16
18
20
```

Another example:

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

```
main()
{
  int n,i;
  cout << “Enter the n value”;
  cin >> n;
  i=1;
    while( i<=n)
      {
       cout << i;
       i++;
      }
}
```

Output :

```
Enter the n value
If the user enters , the output would be
1 2 3 4
```

The update could also be done within the condition itself:

```
#include <iostream.h>
int main(void)
{
   int num = 0;
   while (num++ < 10)
      cout << num << " ";
   return 0;
}
```

Updating the counter within the condition requires two changes from the previous code. First, the value of *num* has to be initialized to 0 instead of to 1 because the increment inside the parentheses during the first iteration of the loop would change that variable’s value to 1. Second, the relational operator in the condition is < rather than <= because the value of *num* is being incremented before it is outputted.

Updating the counter within the condition raises the question: Given the condition *num++ < 10*, which comes first, the comparison or the increment? Since the increment is postfix, the answer is the comparison.

What would be the output if we placed a semicolon after the while condition as in the following code fragment?

```
   while (num <= 10);
      cout << num++ << " ";
```

The only number that would output is 11. The reason is that the loop continues, and the empty statement executes, until the condition fails when *num* is 11, at which time the statement following the loop executes and the value of *num* (11) is outputted.

Nesting While Loops

A program that prints 5 rows of 10 X characters. The following is a modification of that program using nested while loops.

```
#include <iostream>
using namespace std;
int main(void)
{
   int x = 0;
   while (x++ < 5)
   {
      int y = 0;
      while (y++ < 5)
         cout << "X";
      cout << '\n';
   }
   return 0;
}
```

The variable *y,* used as a counter in the inner while loop, needs to be reinitialized in the outer while loop. The variable *y* could be declared outside the loops, but it needs to be assigned (or reassigned) the value of zero inside the outer loop since the inner loop goes through all of its iterations for each iteration of the outer loop.

Since each loop has a predictable number of iterations, using nested for loops is somewhat simpler than using nested while loops. However, both work.

Quiz:

1. int I;

   While (I < 10)

```
{ cout << I << endl;
   I++; }
```

What should I be initialized so that the loop would be printed out 1 to 10?

a. 0
b. 1
c. -1

2.
```
int x = 1, total;
While ( x<=10){
    total + = x;
    ++x;
Cout<< Total;
```

Total =?

a. 10
b. 55
c. Error

3.
```
int c = 1, product;
While( c <= 5)
{ product * = c;
    ++c;
    Cout<<product;}
```

The output of the previous code would be as

a. 5 25 125 625 3125

b. 1 2 6 24 120

c. 2 8 16 24 48

3.12 Do While loop Structure

The do…while loop guarantees that the statement or statement block is executed at least once i.e the condition is tested after execution. The loop is exit-controlled (bottom-tested). For as many iterations as the condition evaluates to true, the statement block is executed.

The syntactic form of do-while loop is

```
do
{
statements;
} while ( condition );
```

The sequence is as follows

- Executes the statement
- Evaluate the condition. If the condition is true, repeat step1

  Since the test condition is evaluated at the bottom of the loop, body of the loop is always executed at least once.

The flow chart makes it clear that the loop continuation condition is not executed until after the action is performed at least once.

Algorithm:

```
counter = 2
Do
Print counter
counter = counter + 2
Loop While counter <= 20
```

C++ code snippet

```
. . .
int counter;
. . .
     counter = 2;
          do
              {
                cout << counter;
                cout << endl;
                counter = counter + 2;
              } while (counter <= 20);
. . .
```

The output would be:

```
2
4
6
8
10
12
14
16
18
```

```
20
```

Another Example:

```
#include <iostream.h>
main()
{
  int n,i;
  cout << “Enter the n value”;
  cin >> n;
  i=1;
    do
      {
       cout << i;
       i++;
      } while (i <=n);
}
```

Output :

```
Enter the n value
(If the user enters , the output would be)
1 2 3 4
```

Using while loop to continue to prompt the user to enter a positive number until the user either did so or quit, and then either outputted the positive number or a message that the user did not enter a positive number.

```
#include <iostream.h>
int main(void)
{
   int num;
   char choice;
   bool quit = false;
   do {
```

```
      cout << "Enter a positive number: ";
      cin >> num;
      if (num <= 0)
      {
      cout << "Number must be positive; try again (Y/N): ";
      cin >> choice;
      if (choice != 'Y')
      quit = true;
      }
   } while (num <= 0 && quit == false);
   if (quit == false)
      cout << "The number you entered is " << num << " ";
   else
      cout << "You did not enter a positive number";
   return 0;
}
```

The following are sample inputs and outputs. The first one has the user successfully enter a positive number the first time.

```
Enter a positive number: 4
The number you entered is 4
```

The next sample input and output has the user enter a positive number after two unsuccessful tries.

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter a positive number: -1
Number must be positive; try again (Y/N): Y
Enter a positive number: 4
The number you entered is 4
```

The final sample input and output has the user quit after two unsuccessful tries.

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter a positive number: -1
Number must be positive; try again (Y/N): N
You did not enter a positive number
```

Quiz:

1. What would be the output of the following segment of code?

   int I;

   I = -12;

   do { cout << I << endl;

I=I – 1 ;}

While (I >= 0)

a. -12

b. -12 -11

c. -12 . . . 0

2. What would be the output of the following segment of code?

int I;

I = -12;

do { cout << I << endl;

I=I – 1 ;}

While (I < 0)

a. -12 . . . 0

b. -12 . . . 1

c. Infinite loop

3. What would be the output of the following segment of code?

int counter=2;

do { cout << counter << endl;

counter + = 2 ;}

While (counter < 100)

a. 2 4 6 . . . 98

b. 2 4 6 . . . 100

c. 0 2 4 . . . 100

3.13 Break Statement

A **break** statement terminates the nearest enclosing *while, do-while, for* or *switch* statement. Execution resumes at the statement immediately following the terminated statement.

Syntax

```
break;
```

Example 1:

Program to demonstrate the use of break statement.

```
#include <iostream.h >
main()
{
  int n,i;
  cout << “Enter the n value”;
  cin >> n;
    for (i=1; i<=n; i++)
      {
        cout << i;
          if ( i= = 3 )
        break;
      }
}
```

Output

```
Enter the n value 5

1 2 3
```

Example 2:

The following modification of the program uses the break keyword to provide the user with the option of quitting the data entry:

```
#include <iostream.h>
int main(void)
{
   int num;
   char choice;
   cout << "Enter a positive number: ";
   cin >> num;
   while (num <= 0)
   {
      cout << "Number must be positive; try again (Y/N): ";
      cin >> choice;
      if (choice == 'Y')
      {
         cout << "Enter number: ";
         cin >> num;
      }
      else
         break;
   }
   cout << "The number you entered is " << num << " ";
   return 0;
}
```

Here is some sample input and output when the user eventually enters a positive number:

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter number: -1
Number must be positive; try again (Y/N): Y
Enter number: 3
The number you entered is 3
```

Here is some sample input and output when the user does not enter a positive number but instead decides to quit:

```
Enter a positive number: -2
Number must be positive; try again (Y/N): N
```

The number you entered is -2

Quiz:

1. What would be the output of the following segment of code?

int power2 = 1;

for(int i =0;i<10;i++)

{ power2=i* 2;

If (power2 % 2)

Break;

Else

Cout<<power2;}

a. 024681012141618
b. 0
c. Nothing

2. What would be the output of the following segment of code?

int power2 = 1;

while(power2<10){

If (power2 % 2)

Break;

Else

Cout<<power2<<" ";

Power2=power2* 2;}

a. 1 2 4 8
b. 1
c. Nothing

Control Structures in C++

3.14 Continue Statement

A continue statement causes the current iteration of the nearest enclosing loop statement to terminate. Execution resumes with the evaluation of the condition. Continue terminates only the current iteration. The continue statement can legally appear only within a loop statement.

Syntax

```
continue;
```

Example 1:

Program to demonstrate the use of continue statement.

```
#include <iostream.h >
main()
{
   int n,i;
   cout << “Enter the n value”;
   cin >> n;
      for (i=1; i<=n; i++)
         {
          cout << i;
            if ( i= = 3 )
          continue;
          cout << i;
         }
}
```

Output

```
Enter the n value
(assume the user enters 4)
1 2 4
```

Example 2:

A program in which the user is charged $3 an item, but not charged for a “baker’s dozen,” so every 13th item is free—that is, the user is only charged the price for a dozen items, even though they receive 13. The following is a modification of that program using a while loop.

```
#include <iostream.h>
int main(void)
```

```
{
   int num, counter = 0, total = 0;
   cout << "How many items do you want to buy: ";
   cin >> num;
   while (counter++ < num)
   {
      if (counter % 13 == 0)
         continue;
      total += 3;
   }
   cout << "Total for " << num << " items is $" << total;
   return 0;
}
```

```
Output
How many items do you want to buy: 14
Total for 14 items is $39
```

Note The % (modulus) operator is used if the remainder is 0, 13, or a multiple of 13 items.

Quiz:

1. What would be the output of the following segment of code?

int power2 = 1;

for(int i =0;i<10;i++)

{ power2=i* 2;

If (power2 % 2)

Continue;

Else

Cout<<power2;}

a. 024681012141618
b. 0
c. Nothing

2. What would be the output of the following segment of code?

int power2 = 1;

```
while( power2<10){

If (power2 % 2)

continue;

Else

Cout<<power2<<" ";

Power2=power2* 2;}
```

a. 1 2 4 8
b. 1
c. Infinite loop

Control Structures in C++

Summary

- Different C++ operators are Logical Operators, Relational Operators and Conditional Operator.
- In this chapter we have seen the various control statements available in C++.
- The control flows from one statement to the next in the sequence structure.
- Conditional branching takes place in the selection structures.
- The ***If*** structure performs an indicated action only when the condition is True; otherwise the action is skipped.
- The ***If…else*** structure allows programmers to specify that a different action is to be performed when the condition is *True* than when the condition is *False.*
- We have applied the ***If*** and ***If…else*** selection structures to choose between alternative action statements.
- ***Nested If*** structures test for multiple cases by placing ***If*** structures inside other ***If*** structures.
- We have then applied multiple selection using the ***Switch…Case*** multiple selection structure.

- There are 3 different looping constructs available in C++ - ***for*** loop, ***do*** loop and ***while*** loop.

- The ***for*** loop has 3 distinct parts – the *initialization* part, the *condition* part and the *increment / decrement* part.

- The ***while*** loop is a *pre-tested* loop – the body is entered only when the condition is satisfied.

- The ***do*** loop is a *post-tested* loop – the body is entered first without the condition being tested, and the check happens only at the end.

- The ***break*** statement takes control out of the current structure.

- The ***continue*** statement takes control back to the condition in the looping structure.

******* PART 1*******

CH.5
ARRAYS

***** PART 2*****

INDEX

***** PART 3*****

LEARNING OBJECTIVES

******* PART 4*******

4.1 The Concept of Array

* An **Array** is a consecutive group of memory locations that all have the same name and the same data type.

* To refer to a particular location or element in the array, specify the name of the array and the position number of the particular element in the array. For instance, in the following figure, the first element can be referred as M[0], and the second element can be referred as M[1], and so on

* The first element in every array is the zeroth element. Thus, if we have an array **A** of **n** elements, the last element can be referred as **A[n-1]**.

* Array names follow the same conventions as other variable names

* While the complete collection or set of variables is called an **array**, the individual values in the array are called **elements**. The position number contained within square brackets is called **index** or **subscript**.

*******PART 5*******

4.2 One Dimensional Arrays

* Arrays can be classified into two main types:

- One-Dimensional Arrays (1-D Array)
- Multi-Dimensional Arrays (N-D Array)

1) Declaring 1-D Arrays

* The general form of a 1-D array is:

DataType arrayName [arraySize]

* The following statements represent an example to declare three arrays:

Example:

```
int   A[10];    // Declare an array called A of 10 integers
float  arr[5];  // Declare an array called arr of 5 float numbers
char  Z[30];  // Declare an array called Z of 30 characters
```

* More than one array of the same data type can be declared in one statement as in the following example.

Example:

```
int  A[10], B[18];
```

2) Initializing 1-D Arrays Using the Initializer List

The elements of an array can be initialized by following the declaration with an equals sign and a comma-separated list enclosed in braces as in the following example

Example:

```
int  a[5] = {3, 7, 4, 9, 2};
float  n[4] = {2.2, 4.9, -9.2, 3};
```

3) Examples Using 1-D Arrays

Ex1

(Declaring array, initializing the elements using loop, and printing):

```
int a[5];
for (int i = 0; i < 5; i++)
{
  a[i] = 1;
  cout<< a[i] << ”\t”;
}
```

Output:

1 1 1 1 1

Ex2

(Declaring array using initializer list and printing):

```
int b[7] = {2, 5, 1, 9, 9, -3, 4};
for (int i = 0; i < 7; i++)
  cout<< b[i] << ”\t”;
```

Output:

2 5 1 9 9 -3 4

Ex3

(Declaring array, input the elements from the user and printing):

```
int arr [10];
for (int i = 0; i < 10; i++)
{
cout<< ”Enter element ” << i << “:” ;
cin>> arr[i];
}
for (int j = 0; j < 10; j++)
cout<< arr[j] <<”\n”;
```

Output:

The output depends on the inputs entered by the user. Try to test this program

Ex4

(Giving a fewer initializers than elements in the array and printing):

```
int  a [7] = {3, 8};
for (int j = 0; j < 7; j++)
cout<< a[j] <<"\t";
```

Output:

3 8 0 0 0 0 0

Note: If there are fewer initializers than elements in the array, the remaining array elements are initialized to zero.

Ex5

(Giving more initializers than elements in the array):

```
int  b [4] = {9, 2, 4, 3, 8};
for (int j = 0; j < 4; j++)
cout<< b[j] <<"\t";
```

Output:

This program will produce a syntax ERROR because there are more initializers than elements in the array.

Ex6

(Declaring an array without initializing or assignment, and printing):

```
int m[10];
for (int j = 0; j < 10; j++)
cout<< m[j] <<"\t";
```

Output:

This program prints ten *rubbish data* because the array has not been initialized absolutely.

Ex7

(Omitting array size with initializer list):

```
int  a []={2, 7, 9, 1};
for (int j = 0; j < 4; j++)
cout<< a[j] <<"\t";
```

Output:

2 7 9 1

Ex8

(More ideas):

```
int D[7] = {2, 7, 5};
D[1] = 9;
D[3] = 8;
D[5] = 7;
for (int j = 0; j < 7; j++)
cout<< D[j] <<"\t";
```

Output:

2 9 5 8 0 7 0

Ex9

(Initializing array elements with calculations):

```
int Z[8];
for (int i = 0; i < 8 ; i ++)
{
  Z[i] = i * i ;
  cout<< Z[i] << "\t" ;
}
```

Output:

0 1 4 9 16 25 36 49

Ex10

(Summing the elements of array):

```
int  marks [10] = {91, 88, 71, 84, 66, 99, 69, 79, 81, 71};
int sum = 0;
for (int i = 0; i < 10 ; i ++)
    sum += marks[i];
cout<< "The Total = " << sum;
```

Output:

The Total = 799

*******PART 6*******

4.3 TWO Dimensional Arrays

The following figure represents a 2-D array called **A**

To identify a particular table element, we must specify two subscripts: The first identifies the element's row, and the second identifies the element's column.

1) Declaring 2-D Arrays

* The general form of a 1-D array is:

DataType arrayName [NumberOfRows][NumberOfColumns]

* The following statements represent an example to declare three 2D arrays:

Example:

```
int   A[2][3];      // Declare an array called A of size 2*3 integers
float  arr[4][3];  // Declare an array called arr of 4*3 float numbers
char  Z[5][4];   // Declare an array called Z of 5*4 characters
```

* More than one 2D array of the same data type can be declared in one statement as in the following example.

Example:

```
int  A[3][5], B[2][5];
```

2) Initializing 2-D Arrays Using the Initializer List

The elements of an array can be initialized by following the declaration with an equals sign and a comma-separated list enclosed in braces. The values are grouped by row in braces as in the following example

Examples:

```
int  a[2][3] = {{3, 7, 4},{ 9, 2, 5}};
float  n[3][3] = {{2.2, 5.1,4.9}, {-9.2, 3, 2.3},{4.1, 4.2, 4.5}};
int d [2][2] = {3, 5, 1, 6};
```

- Notice that the values in array d are not necessary to be grouped by row

3) Examples Using 2-D Arrays

Ex1

(Declaring array, initializing the elements using loop, and printing):

```
int a[2][3];
```

```
for (int r = 0 ; r < 2 ; r++)
{
    for (int c = 0; c < 3 ; c++)
   {
   a[r] [c] = 1;
   cout<< a[r][c] << ”\t”;
   }
cout << ”\n”;
}
```

Output:

```
1       1       1
1       1       1
```

Ex2

(Declaring array using initializer list and printing):

```
int b[2][2] = {2, 5, 1, 9};
for (int r = 0 ; r < 2 ; r++)
{
    for (int c = 0; c < 2 ; c++)
   {
   cout<< a[r][c] << ”\t”;
   }
cout << ”\n”;
}
```

Output:

```
2       5
1       9
```

Ex3

(Declaring array, input the elements from the user and printing):

```
int arr [3][2];
for (int r = 0 ; r < 3 ; r++)
    for (int c = 0; c < 2 ; c++)
   {
cout<< ”Enter element ( ” << r << “,” << c << “):” ;
cin>> arr [r] [c] ;
  }
```

```
for (int r1 = 0 ; r1 < 3 ; r1++)
{
    for (int c1 = 0; c1 < 2 ; c1++)
   {
   cout<< a[r1][c1] << ”\t”;
   }
cout << ”\n”;
}
```

Output:
The output depends on the inputs entered by the user. Try to test this program

Ex4
(Giving a fewer initializers than elements in the array and printing):

```
int  a [2][3] = {2, 5};
for (int r = 0 ; r < 2 ; r++)
{
    for (int c = 0; c < 3 ; c++)
   {
   cout<< a[r][c] << ”\t”;
   }
cout << ”\n”;
}
```

Output:

```
2       5       0
0       0       0
```

Note: If there are fewer initializers than elements in the array, the remaining array elements are initialized to zero.

Ex5
(Giving more initializers than elements in the array):

```
int  b [2][2]= {7, 2, 5, 3, 8};
```

Output:
This program will produce a syntax ERROR because there are more initializers than elements in the array.

Ex6

(Declaring an array without initializing or assignment, and printing):

```
int m[2][3];
for (int r = 0 ; r < 2 ; r++)
{
    for (int c = 0; c < 3 ; c++)
   {
   cout<< a[r][c] << ”\t”;
   }
cout << ”\n”;
}
```

Output:

This program prints six *rubbish data* because the array has not been initialized absolutely.

Ex7

(More ideas):

```
int D[2][3] = {2, 7, 5, 1};
D[0][1] = 9;
D[1][0] = 8;
for (int r = 0 ; r < 2 ; r++)
{
    for (int c = 0; c < 3 ; c++)
   {
   cout<< D[r][c] << ”\t”;
   }
cout << ”\n”;
}
```

Output:

```
2       9       5
8       0       0
```

Ex8

(Initializing array elements with calculations):

```
int Z[3][3];
for (int r = 0 ; r < 3 ; r++)
{
```

```
    for (int c = 0; c < 3 ; c++)
   {
     Z[r][c] = r  + c ;
     cout<< Z[r][c] << ”\t”;
   }
cout << ”\n”;
}
```

Output:

```
0       1       2
1       2       3
2       3       4
```

Ex9

(Summing the elements of array):

```
int  marks [3][3] = {91, 88, 71, 84, 66, 99, 69, 79, 81};
int sum = 0;
for (int r = 0 ; r < 3 ; r++)
{
    for (int c = 0; c < 3 ; c++)
   {
sum += marks[r][c];
   }
}
cout << ”Total = ” << sum;
```

Output:

```
The Total = 728
```

CH.5
Character and String Processing

******* PART 2*******

INDEX

***** PART 3*****

LEARNING OBJECTIVES

***** PART 4*****

5.1 Introduction

* A character constant is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in ASCII code.

Eg: char m1 = ‘D’;

The previous statement stores 68 in the variable m1

* A string is a series of characters treated as a single unit. So it may include letters, digits and various special characters such as $, * and +.

* String literals, or string constants are written in double quotation marks.

* A string is an array of characters ending in the null character('\0'), which specifies where the string terminates in memory.

* A string may be assigned in a declaration to a character array as in he following statement:
Eg: char university [ ] = "Balqa";
Notice that the previous statement creates a six-element array containing the characters 'B', 'a', 'l', 'q', 'a' and '\0'.

* The statement:
char university [ ] = "Balqa";
can also be written as:
char university [ ] = {'B', 'a', 'l', 'q', 'a', '\0'};

* When declaring a character array to contain a string, the size of the array must be large enough to store the string and its terminating null character. So it is an error to not preserve an enough space in a character array to store the null character.

* A string can be read from the user and stored in an array using cin. But using cin reads characters until a whitespace character is encountered.
Eg:
```
char arr[40];
cin>>arr;
cout<<arr;
```

Notice that if the user enters balqa university, the characters before the space will be stored in arr and the rest will be ignored). So the program will print balqa.

* C++ provides the function cin.getline to input an entire line of text into an array
  - cin.getline is exist in string.h

- cin.getline can take two parameters: a character array in which the line of text will be stored and a length.

Eg:

```
char arr[40];
cin.getline(arr,25);
cout<<arr;
```

Notice that if the user enters balqa university, all the characters will be stored in arr. So the program will print balqa university.
This function will stops storing characters when '\n' is encountered or when the number of characters read so far is one less than the length specified in the second argument. Remember that the last character in the array is reserved for the terminating null character

******* PART 5*******

5.2 String Functions

* There are various string-manipulation functions that are exist in string.h

* In this section, we will present to you some common functions of them.

1) strlen(s)

Function *strlen* returns the number of characters in the string which takes it as an argument. Notice that the terminating null character is not included in the length

Eg:

```
char m[] = "Salam every one";
cout<<"Number of characters = "<<strlen(m);
```

Output

Number of characters = 15

Eg:

Suppose that the user enters (C++ exam), what is the output??

```
char m[30];
cin.getline(m,25);
cout<<"Number of characters = "<<strlen(m);
```

Output

Number of characters = 8

2) strcpy(s1, s2)

Function *strcpy* copies its second argument into its first argument (which must be large enough to store the string and its terminating null character)

Eg:

```
char m[] = "Salam every one";
char z[30];
strcpy(z,m);
cout<<m<<"\n"<<z;
```

Output

Salam every one
Salam every one

Eg:

```
char m[] = "Salam every one";
char z[30]="Welcome";
strcpy(z,m);
cout<<m<<"\n"<<z;
```

Output
Salam every one
Salam every one

3) strcat(s1, s2)

Function *strcat* appends its second argument to its first argument. The first character of the second argument replaces the null character that terminates the first argument. The programmer must ensure that the array used o store the first string is large enough to store the combination of the first string, the second string and the terminating null character copied from the second string.

Eg:

```
char m[20]="Salam";
char a[20]=" every one";
strcat(m,a);
cout<<m<<"\n"<<a;
```

Output
Salam every one
every one

4) strcmp(s1, s2)

Function *strcmp* compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a negative value (-1)if the first string is less than the second string and a positive value (+1)if the first string is greater than the second string.

Eg:

```
char m[]="Apple";
char a[]="Table";
char z[]="Apple";
char b[]="TableAndChair";

cout<<strcmp(m,a)<<"\n"<<strcmp(a,m)<<"\n";
cout<<strcmp(m,z)<<"\n"<<strcmp(a,b);
```

Output
-1

1
0
-1

6.1 Introduction

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

The fundamental idea of a function in C++ is that of a set of actions bundled together and given a name

It emphasize the idea of code reusability, instead of writing a piece of code several times, we write it once and use it many.

main()is a function, and a program needs exactly one main()function.

It is not recommended to have lengthy main()function, a better way to manage your code and make more traceable, readable, and correctable if errors happen is to split it into smaller pieces (functions), each of those functions performs a specific task

So far, we have introduced and used functions defined in header files, such as the *pow* function in the *math* library, that is used to raise a number to a certain power, and then getting the result

Another example of introduced functions, strcpy function in the string library, which is used to copy a string into another one.

Functions are either user- defined or predefined.

Examples of Predefined functions pow(), sqrt(), strlen(), strcmp(), strcpy()

Notice that functions have input/inputs, output, and functionality or job that it achieves.

For example pow() takes two inputs, and they are of type double, the input is called parameter. So, pow() takes two parameters, its functionality is to find the result of the first power the second. And it gives this result back each time pow function is used

Calling a Function

Using a function is called in programming calling a function

To call a function you have to name it, give it the right parameter/s (number of parameter/s and type of parameters), and put it in the right context/place

Following are examples of calling pow() function (suppose math.h is included):

```
1  Int x=9, y=4;
2  double z=3,w;
3  cout<< pow(x,4);
4  w=pow(y,x-4);
5  w=4*pow(2+y,z);
6  pow(2.5,2);
```

The first call of pow will compute xy and the cout will display it on the output screen.

6.2 Building user-defined functions

Before attempting to build a function, you need a clear idea of what it is that you are trying to do

Building any function needs two steps:

1. function prototype
2. function definition

Function prototype:

Function prototype is a brief description of a function, it gives a general idea about the function, what kind of data it will return, what it is called, and what type of input it takes.

Function input/s is called parameter/s

The syntax of the function prototype is:

Return data type functionName(list of parameters datatypes);

- Return data type: is the data type your function is expected to return.
    - The function has only one return data type, and in case of not returning anything use void.
- functionName is the chosen name of the function, and it follows the same rules of variable naming in c++
- List of parameters data types, it is a comma separated list, containing the date types of the parameters (input), all enclosed in braces.

    - If the function takes no input use void or leave the braces empty, otherwise for each input mention its data type.
- The prototype must end with a semicolon.

The location of the function prototype is just before main() function.

The prototype is above main(). This ensures that the compiler, compiling the code from top to bottom, will encounter the prototype before any function

Function definition:

Function definition is a detailed description of a function, so in addition to all description given by prototype, it gives deep idea about the job of a function, and how it works.

Function definition includes all information mentioned in prototype.

Function definition syntax:

Return data type functionName(list of parameters definitions)

{

Function code

}

- Return data type: same as in prototype.
- functionName: same in prototype.

- list of parameters definitions : here not only the parameters data types must be mentioned, but also names of each one to handle it within the function code.
- The function code consists of one or more statements.
    - If the Return data type is not void, at least one statement must be :

      return exp;

      Note that the *exp* type must match the Return data type .

    - If the Return data type is void. There is no need for return statement, or return; can be used, note that no exp is following return statement in case of void return data type.
- So, the function body must contain a return statement unless the return type is void, in this case the return statement is optional.

The location of function definition is after the end of main().

A function cannot execute until it is defined.

A function executes when it is *called.*

A function is called through code, and the function call happens in another function, mostly in main().

The next examples will explain the whole process of how to define your function and then call it.

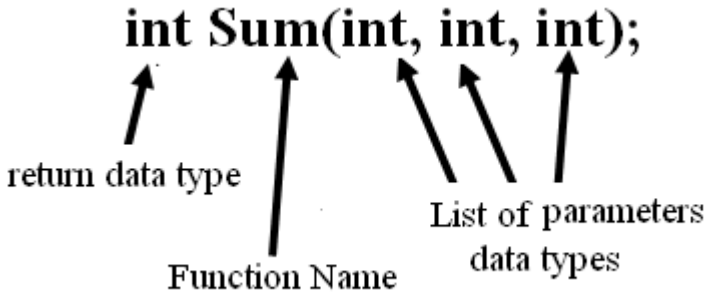
Examples:

Let us start with examples about prototypes only:

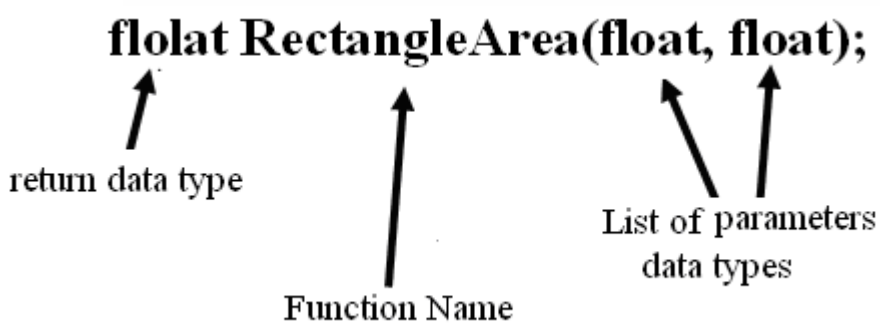
Write a prototype for each of the following:

- **Function that computes the sum of three integers**

The function will take three integers as parameters (inputs), and then it must give the sum as a result which is integer also, the final thing is to choose a function name, let it be Sum, and so by gathering all together we will have:



Let us repeat it with another function, suppose we want to build a function that computes the space of a rectangle, so it needs two inputs (width, length), both of them can be float, so the area also will be a float, finally the easiest part, the function name, let it be RectangleArea, putting it all together will be:



Now, let us continue building the function, and let us continue with Sum, the function definition will like this:

int Sum(int a, int b, int c)

```
{

int s;

s=a+b+c;

return s;

}
```

The return data type and function name are the same as in prototype, the list of parameters data types is replaced by list of parameters definition, in it we give each parameter a name so we can use it to refer to that parameter. Then finally the code, an integer called s is declared, then it is assigned to the sum of given parameters a, b, and c. after that the needed result the function aims to find is saved in s, so we end our Sum function with return s statement.

Let us put both prototype and definition of Sum in a program and call it to see what happens exactly

```
//O. S. AlMousa

#include<iostream.h>

int Sum(int, int, int);

void main()

{

    int w,x=4,y=15,z=-6;

    cout<<"this program is to demonstrate functions\n";
```

```
    w=Sum(x,y,z);
    cout<<Sum(x+5,y-6,10)<<endl;
    cout<<"w = "<<w<<endl;
    cout<<Sum(123, Sum(w,4,7),y*-1)<<endl;
}


int Sum(int a, int b, int c)
{
int s;
s=a+b+c;
return s;
}
```

Firstly, four variables are defined, namely w,x,y, and z with the values 9,4, 15, and -9 respectively.

cout will display “This program is to demonstrate functions”

and then the first call to Sum happens, this call will cause Sum to execute, so the values of the parameters will be saved in the parameters list respectively, i.e, the value of ***x*** will be saved in ***a***, so ***a*** will equal to 4, and so will happen with the rest, so the value of ***b*** will be 15, and the value of ***c*** will be -9.

After that an integer ***s*** will be declared, the next statement in Sum will save the value of ***a***+***b***+***c*** in ***s.*** The final statement in Sum will take the value of ***s*** back where it was called, and then it will be saved in ***w***, so ***w*** value will be 13.

line 4 in main() will call Sum function again, this time with the values of ***x+5***,***y-6***, and **10**, they will be saved in ***a***, ***b***, and ***c***, again an integer ***s*** will be declared the next statement in Sum will save the value of ***a***+***b***+***c*** in ***s***, and the final statement in Sum will take the value of ***s*** back where it was called, and then it will be given to cout, , so cout will display **28.**

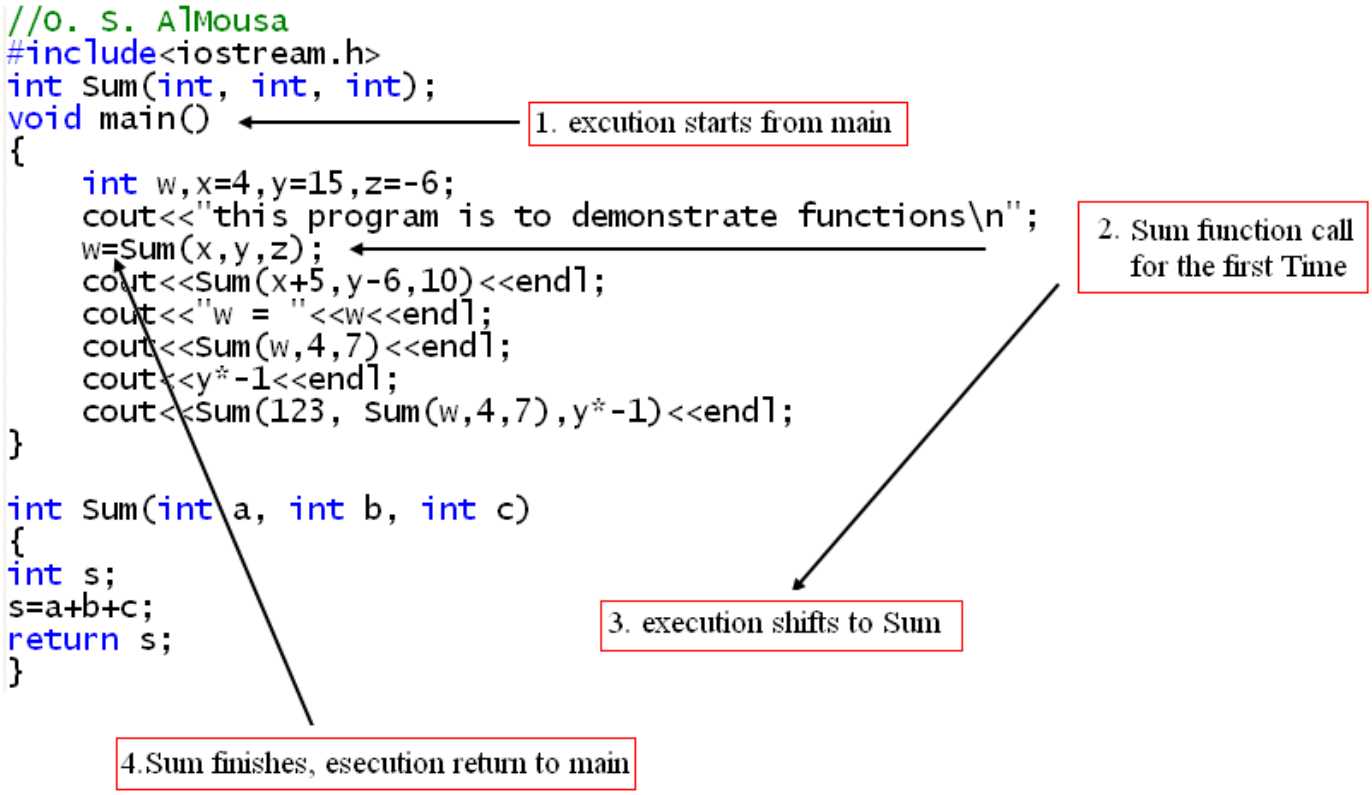
The next line is ordinary, it will display the value of w, the line 6 another call to Sum will happen, in this call the fisrt parameter will be **123**, the second one will be **Sum(w,4,7)**, and the finale one will be ***y*-1,*** so before starting with those parameters a call to Sum (inner) will happen with the values ***w,4,***and ***7***, and they sill be saved in ***a, b,*** and ***c*** respectively, the result will be **24** and it will be used as second parameter in the outer call of Sum, so the outer call of sum will be with the values ***123***, ***24***, and ***y*-1***, and saved in ***a***,***b***, and ***c*** respectively, the result will be calculated and returned to cout statement, which will display it, and it will be equal to ***132***

The final output of the program will be:

```
this program is to demonstrate functions

28

w = 13

132

Press any key to continue
```

Try to build your own rectangle area function.

Now, look at this Figure shows the order of execution graphically.



6.3 Variable Scope and Lifetime

Consider our very program for with these modifications:

//O. S. AlMousa

#include<iostream.h>

int G;

int Sum(int, int, int);

```
void main()
{
	int w,x=4,y=15,z=-6;
	cout<<"this program is to demonstrate functions\n";
	w=Sum(x,y,z);
	cout<<Sum(x+5,y-6,10)<<endl;
	cout<<"w = "<<w<<endl;
	cout<<Sum(123, Sum(w,4,7),y*-1)<<endl;
}


int Sum(int a, int b, int c)
{
int s;
s=a+b+c;
return s;
}
```

Two statement were added, they are bolded for clarity reasons, the first in main( ), cout<<s . And the other in Sum( ), w=12 . and unsurprisingly, they both cause the same error, namely undeclared identifier. This means than variables declared in main( ) are not so in

Sum( ), and vice versa, this is called variable scope, a variable scope is the place where a variable can be accessed or (known). s is local variable in Sum( ), and w is local variable in main( ).

Generally speaking:

Variables: ***w,x,y,z*** are local variables in main( )

Variables: ***a,b,c,s*** are local variables in Sum( )

Local variables are accessed only in their functions, so they have function scope.

The opposite type of variables is global variables, and you may wonder about the place they are declared in, the next code declare a global variable called G**:**

```
//O. S. AlMousa

#include<iostream.h>

int G;

int Sum(int, int, int);

void main()

{

    int w,x=4,y=15,z=-6;

    cout<<"this program is to demonstrate functions\n";

    G=7;

    w=Sum(x,y,z);
```

```
    cout<<Sum(x+5,y-6,10)<<endl;
    cout<<"w = "<<w<<endl;
    cout<<Sum(123, Sum(w,4,7),y*-1)<<endl;
    cout<<G;
}

int Sum(int a, int b, int c)
{
int s;
++G;
s=a+b+c;
return s;
}
```

The place is outside all functions, and before them all.

Notice that ***G*** is accessed in both functions with no problems at all, that is because a global variable is declared for all functions in a program,

It is said that ***G*** has program scope

Global variables have program scope

The last statement in main( ), cout<<***G*** . will display: 11, because ***G*** is initialized to 7, and each time Sum( ) is called 1 is added to ***G***, knowing that Sum( ) was called 4 times, so ***G*** will be after all those calls 11.

Another issue is variable lifetime; variable lifetime is the period a variable lasts in the memory before it is deleted.

Local variables has function lifetime, i.e. whilst the function is executing, all it local variables are alive in memory.

Global variables has program lifetime. i.e they are alive all the program time.

```
//O. S. AlMousa

#include <iostream.h>

void Fun(void);


int main ()

{

   cout<<"The first call of Fun()\n";

   Fun();

   cout<<"The second call of Fun()\n";

   Fun();

   cout<<"The third call of Fun()\n";

   Fun();
```

```
   cout<<"End of Program\n";
    return 0;
}
void Fun (void)
{
   int counter = 0;
   counter++;
   cout << "This function called "<< counter
<<"times\n";
}
```

And following is the output:

```
The first call of Fun()

This function called 1times

The second call of Fun()

This function called 1times

The third call of Fun()

This function called 1times

End of Program

Press any key to continue
```

After compilation, the output is not exactly what we wanted. The variable ***counter*** in the Fun( ) function does not "remember" its previous times that function was called. This is due to being a local variable that is "born" every time its function is called, and the "dies" every time its functions terminates.

So ***counter*** is created each time Fun ( ) is called, and each time it is created it is given the value 0.

The variable ***counter*** is local to the Fun( ) since it was declared in that function. A local variable, each time the Fun( ) is called, the variable ***counter*** is created, and each time the Fun( ) ends, the variable ***counter*** is destroyed. Accordingly, the variable ***counter*** the second time the

Fun( ) is called is not a continuation of the variable ***counter*** that was created the first time the Fun( ) was called. Rather, the variable ***counter*** starts all over again each time the Fun( ) is called.

There are two alternative methods to having the value of a variable remains between function calls. One is to make the variable global rather than local. The other is to keep the variable local but make it static, this one will be covered next

6.4 Static Local Variables

A static local variable has the scope of a local variable but the lifetime of a global variable.

The following program will demonstrate the concept:

```
//O. S. AlMousa

#include <iostream.h>

void Fun(void);


int main ()

{

    cout<<"The first call of Fun()\n";

    Fun();

    cout<<"The second call of Fun()\n";
```

```
    Fun();
    cout<<"The third call of Fun()\n";
    Fun();
    cout<<"End of Program\n";
    return 0;
}
void Fun (void)
{
  static int counter = 0;
  counter++;
  cout << "This function called "<< counter <<"times\n";
}
```

And here is the output:

```
The first call of Fun()
This function called 1times
The second call of Fun()
This function called 2times
The third call of Fun()
```

This function called 3times

End of Program

Press any key to continue

This output also is what we wanted.

The first time the Fun( )is called, the variable ***counter*** is declared, and initialized to zero, by the statement:

```
static int conuter = 0;
```

The variable *conuter* then is incremented and outputted, resulting in the output:

```
This function called 1 times
```

So far, this is the same as when *times* was a local variable rather than a static local variable.

The difference is that when the Fun( )ends, *conuter*, being a static local variable, is *not* destroyed.

That variable and its value remain in memory.

The next (second) time the *Fun( )*is called, the statement declaring and initializing variable *conuter* is *not* executed because that variable, being static, still exists from the first time the *Fun( )*was called. Accordingly, the value of the *conuter* variable at the end of the first call of the Fun( ), 1, remains in memory. That value then is incremented to 2, and outputted, so the output to the second call of the Fun( )is

```
This function called 2 times
```

The next (third) time the *Fun( )*is called, the statement declaring and initializing variable *conuter* is ignored again, and the value of *conuter* at the end of the second call of the Fun( ), 2, remains in memory. That value then is incremented to 3, and outputted, so the output to the third call of the Fun( ) is

```
This function called 3 times
```