1 INTRODUCTION

1.1 DEFINITIONS

1.1.1 PROGRAM

A program is a set of instructions written in a language that a computer can understand. Programs are normally used to instruct a computer to do something.

1.1.2 COMPILER

A compiler is a program that translates a high level language such as C++ program into a machine language program that the computer can directly understand and execute. Therefore a compiler is a program whose input data is a program and its output is yet another program. To avoid confusion the input program is usually called the source **program or source code** and the translated program the **object program or object code.**

1.1.3 SOURCE CODE

The program written by the programmer using a high level language e.g. C++.

1.1.4 OBJECT CODE

The machine language translation of a program written using a high level language produced by the computer when a program is compiled.

1.1.5 ALGORITHM

A sequence of precise instructions that lead to a solution.

1.2 PROGRAM / SOFTWARE LIFE CYCLE.

The process of developing a computer program generally involves six stages which are referred to as the program development life cycle and are shown below diagrammatically.

1.3 STEPS

1.3.1 REQUIREMENT SPECIFICATION

Problem specification state the purpose / function of the program. It involves the definition of inputs, transformations, outputs and treatment of exceptional situations.

1.3.2 PROGRAM DESIGN

This is whereby the programmer identifies the processing tasks required and the precise order in which they are to be carried out, that is the algorithm from which the computer program will be produced. The design process takes little or no account of the programming language to be used. In the final product since the emphasis o\is on defining program logic at this stage.

1.3.3 CODING THE PROGRAM

After having designed the logical structure of the program, the programmers next stage is to convert the design specification (the algorithm) into actual computer instructions in the computer language to be used. If sufficient care has been taken at the design stage, coding is an almost mechanical process for an experienced programmer since most of the hard work of analyzing the problem and breaking it into a sequence of small, simple steps has already been done during the design stage.

1.3.4 TESTING AND DEBUGGING

Debugging is the detection and correction of errors that may exist in the program. The only way to be completely confident that a program will behave as expected, under all circumstances that conceivably arise, is to test the program thoroughly before implementing it. Program testing includes creating test data designed to produce predictable output.

1.3.5 DOCUMENTATION

Having written and debugged a program it must be documented. There are two main types of documentation.

a. **User Guide** – These are guides that are intended to help the user to use / operate the program with minimal or no guidance.
b. **Technical Manuals –** These are intended for system analysts / programmers to enable maintenance and modification of the program design and code.

1.3.6 IMPLEMENTATION

At this stage the system designers will actually install the new system, create data files and train people to use the new system. A common practice is to run the new system in parallel with the old system to ensure that if something does go wrong, the business does not come to a grinding halt.
After a number of parallel runs, the old system will be abandoned and the new system will be fully operational. At this point the system goes into the final stage of monitoring and review (or evaluation) where the performance of the system is evaluated with reference to the initial requirements.

1.4 QUALITIES OF A GOOD PROGRAM

1.4.1 ACCURACY

The program must do what it is supposed to do. The results / output should be correct as per specifications.

1.4.2 RELIABILITY

The program must always do what it is supposed to do and never crash.

1.4.3 ROBUSTNESS

The program should cope with invalid data without creating errors or stopping without indication of the cause.

1.4.4 USABILITY

To make sure that your program is easy to use, make sure that the flow of your program is logical and provide a proper use user guide / manual.

1.4.5 READABILITY

To make your code make readable use the following rule / guidelines.

a.) **Use Meaningful Data Names** – When coming up with names for variables or constants, use names that hint what the variable or constant holds e.g. balance, tax, tax_rate e.t.c instead of using letters like ab,b,c.

b.) **Use Comments** – Use comments at the module level to explain the purpose of the module and within code to explain complicated algorithms or highlight error prone sections.

c.) **Use Indentation** – Layout your code neatly and make proper use of indentation to reflect the logic structure of the code.

d.) **Modularization** – Split your program to make it easier to understand. In C++ this is normally achieved by including functions in your program. Modularizing a program, reduces the effort required later to change the program since changes need only to be made to the module whose functionality changes or if a new feature is to be added then the new module can be added. Testing and debugging also becomes easier with small independent modules.

1.5 TYPES OF ERRORS

There are two main types of errors found in computer programs.

a. **Syntax Errors** – These are errors in the use of the programming language. every programming language has a set of rules concerning formal spelling, punctuation, naming of variables and other conventions that must be obeyed by the programmer. These set of rules is called syntax and its violation is what is referred to as syntax errors. These errors are detected by the compiler.

b. **Logical Errors** – These are errors in the underlying algorithm of a program or in translating the algorithm into the programming language to be used. For example if you were to mistakenly use the addition (+) sign instead of multiplication (*) sign , that would lead to one of two results:

   i. The program will produce wrong results but will finish normally.

   ii. An usual condition will be encountered (e.g. attempting to divide by zero) that leads to a premature end of the program. This type of error is known as **runtime error**. These are kind of errors that the computer can detect only when a program is run.

1.6 PROGRAM TESTING STAGES

There are several stages in testing the program:

i. Desk checking
ii. Compiler (or assembler) system checking
iii. Program run with test data
iv. Diagnostic procedures
v. Full scale (system) test with actual data

1.6.1 DESK CHECKING (or Dry Run)

After writing the program, the programmer goes through the program on paper to pick up any errors that would cause extra work at a later stage.

1.6.2 TRANSLATOR SYSTEM CHECKING

After keying in the program, it is checked using the translator to detect any syntax errors. The programmer corrects the errors and re-submits the program until an error free listing is obtained.

1.6.3 PROGRAM RUN WITH TEST DATA

Test data is created designed to produce predictable output. The program is then run using this data and its output is compared with the predicted output. Test data should include all important variations and extremes or data including data with errors to ensure that the program does not grind to an halt if incorrect data is read in.

1.6.4 DIAGNOSTIC PROCEDURES

For complex programs diagnostic procedures such as trace routines, may be used to find logical errors. A trace prints results at each processing step to enable errors to be detected quickly. If a trace routine is not available the programmer inserts instructions in the program to print results at key points.

1.6.5 SYSTEM TEST WITH ACTUAL DATA

As part of the “Acceptance Trials” new programs are usually run in parallel with the existing system for a short while so that results can be compared and adjustments made. The system run using actual data.

2 INTRODUCTION TO C + +

C ++ language was derived from the C language. Unlike C C ++ has facilities to do object oriented programming.

2.1 LAYOUT OF A SIMPLE C++ PROGRAM

The layout of a simple C ++ program is shown below

```
#include<iostream.h>
int main()
{
        variable declarations;
        statement1;
        statement2;

        statement last;
return0;
}
```

The line **#include<iostream.h>** is called the **include directive**. It tells the compiler where to find information about certain items that are used in your program. In this case iostream.h is the name of a library that contains the definitions of the routines that handle input from the keyboard and output to the screen **iostream.h** is a file that contains some basic information about this library.
The following lines simply say that the main part of the program starts here
int main()
{

strictly speaking, the correct term is main function rather than main part. The curly braces { and } mark the beginning and end of the "main part" of the program .
The line **return 0;** says "End the program when you get here". This line need not be the last thing in the program but in very simple programs, it makes no sense to place the curly brace anywhere else.
Variable declarations are normally placed as the first statement in the main function. Even though it is not a must you place variable declarations at the beginning of the main function, it is a good default location for them.
The statements are instructions that are followed by the computer e.g. input and output statements, calculations e.t.c. They are sometimes called **executable statements** and should always end with a semicolon.

2.2 VARIABLES

A variable is a named memory location that contains a value that can change during program execution. C ++ variables can hold numbers or other types of data. The number or other data held in a variable is called its value.
Every variable in a C ++ program must be declared. When you declare a variable, you are telling the compiler and ultimately, the computer what kind of data you'll be storing in the variable.
The kind of data that is held in a variable is called its type and the name for the type, such as int or double is called the type name.

SOME NUMBER TYPES

Type Name	***Memory Use***	***Size Range***	***Precision***
int	2 bytes	-32,767 to 32,767	not applicable
long also called long int	4 bytes	-2,147,403647 to 2,147,403,647	not applicable
float	4 bytes	Approximately 10^{-38} to 10^{38}	7 digits
double	8 bytes	Approximately 10^{-308} to 10^{308}	15 digits
long double	10 bytes	Approximately 10^{-4932} to 104^{932}	19 digits

These are just sample values to give you a general idea of how the types differ. The values of any of these entries may differ on different systems. The ranges for types **float, double and long double** have a similar range, but with a negative number input of each number.
The type int is used to hold integers / whole numbers (i.e. numbers without a fraction part) while the types float and double are used to hold a number that contains a fraction part. Numbers of type int are stored as exact values while those of type float and double are stored varies from one computer to another.
The type char (which is short of character) is used for storing a single symbol such as a letter, digit or punctuation mark.

2.2.1 NAMING OF VARIABLES

When naming variables, there are some rules that should be followed. These are:

i. It can be made up of letter, digits and underscore (_).
ii. No other special character are allowed.
iii. First character should be a letter or an underscore (_). It can be followed by letters, digits or underscore.
iv. Uppercase and lower case letters are significant. Both case are allowed. MARKS, marks, and MaRks are considered to be three difference variables.

v. Reserved words cannot be used as variable names.

Keywords / Reserved words are words that have a predefined meaning in C ++ e.g. int, long, float, double, goto, return, if, for, do, while, void, e.t.c.
The following variable names are valid: -
Radius, a, gross_pay, _root, student1, no8
The following variable names are invalid: -

Variable name	**Reason**
7th	first character is a digit
void	void is a reserved word
time?	? is a special character
my salary	it has a space (which is a special character)

2.2.2 VARIABLE DECLARATIONS

Syntax: type_name	Variable list;
Example	Remarks
int a;	a is declared as an integer
int x,y,z;	x, y, z are declared as integers
int m, n = 5, q = 25;	m, n and q have been declared as integer. n and q have been initialized with 5 and 25 respectively.
float basic_sal,net_sal	basic_sal and net_sal have been declared as floating point variables
double allowance;	Allowance has been declared as of type double
char Student	Student is a character variable.

2.3 CONSTANTS

A constant is a symbolic name that holds a value that cannot change during program execution. You can name a constant using the const modifier.
Syntax:

```
const  type_name constant_name = value;
```

Examples:
1. const int MAX_NUMBER = 100;
In this example, a constant called MAX_NUMBER which is of type int has been declared and it has been assigned the value 100.

2. const double PI = 3.142857;
In this example, a constant called PI which is of type double has been declared and it has been assigned the value 3.142857

2.4 LABELS

Labels are mainly used with a statement, so that the statement can be referenced in future during execution. Any C ++ language statement can have label prefix which should be followed by a colon.
Syntax: Identifier:
This identifier is the label name. The rules for naming a label are the same to those of naming a variable. Usually this label is a target of the goto statement.

2.5 EXPRESSIONS

A C ++ expression is a combination of operators, constants, variables and function calls that result in a single value. An expression is always a part of a statement.
Expressions are generally classified as arithmetic, relational and logical expression.

2.6 STATEMENTS

Statements are language constructors that represent a set of declarations or steps in a sequence of actions.

2.6.1 NULL STATEMENTS

A null statement in C ++ is represented by a single semicolon which is the statement delimeter.
Example ; Null statement

2.6.2 EXPRESSION STATEMENT

A C ++ arithmetic statement is a combination of operators, constants, variables and function calls that result in a single value.
Examples:

i. c = a + b / 2;
ii. f = m + n – sqrt (d);
iii. big (a,b,c); - Function Call

2.6.3 BLOCK OF STATEMENTS

In C ++ language, some statements can be grouped together to make a block by giving them within curly braces ({ }). Such blocking is made for specific purposes such as for executing a block of statements repeatedly. A block may gave its own set of declarations and initializations.

Example:
```
{
        h_allowance = 10.0 / 100 * basic;
        t_allowance = 12.0 / 100 * basic;
        gross = basic + h_allowance + t_allowance;
        tax = 12.0 / 100 * gross;
        net = gross – tax;
}
```

2.6.4 LABELED STATEMENTS
Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control but give the provision so that though other statements transfer of control to the labeled statement is possible.

2.6.5 CONDITIONAL CONTROL STATEMENT
The statements through which the flow is controlled are known as control statements.
Example: if, if … else

2.6.6 LOOP CONTROL STATEMENTS
The repeated execution of the same block of statements is possible with loop statements.
Example do … while, while and for

2.7 OUTPUT (Using cout)
The values of variables as well as strings of text may be output to the screen using cout. There may be a combination of variables and strings to be output.
Examples

i. cout<<"Hello There";
This statement tells the computer to output the string Hello There.

ii. cout<<k;
This statement tells the computer to output the value stored in the variable k. For example if the value stored in k is 25 the output on the screen will be 25.

iii. cout<<"The value is";
cout<<k;

Here we have two cout statements. one that outputs the string T*he value is* and another one that outputs the value stored in the variable k. so if for example the value stored in k is 16, the output will be *The value is 16.*
We can combine the two statements to have one cout statement presented as follows.
cout<<"The value is "<<k;
This statement tells the computer to output two items, the quoted string. The value is and the value in k is 20, the output will be. *The value is 20.*

iv. age = 45;
cout<<"I am "<<age<<"Years Old";
This statement tells the computer to output three items the quoted string *I am*. The value in variable age and the string *years old.* So the output will be *I am 45 years old.*

v. a = 10;
   b = 25;
cout<<a<<" + "<<b<<" = "<<(a + b);
The output from this statement will be: - 10 + 25 = 35
If you have a very long cout statement and you want to keep your program lines from running off the screen, you can place it in two or more lines e.g.

```
cout<<"I am "<<my_age<<"Years old"
<<"and my father is "<<his_age
<<"Years old";
```

NOTE
1. There is only one semicolon for each cout statement at the end of the third line in the example above.
2. You should not break quoted string across two lines.

2.8 STARTING NEW LINES IN OUTPUT
To start output in a new line, you can use \n in a quoted string as in the example below.
cout<<"Good Morning \n Kenya";

Output
```
Good Morning
Kenya
```

Alternatively you can start a new line by outputting of endl.
An equivalent way to write the above cout statement is:
cout<<"Good Morning <<endl<<"Kenya";

NB

endl should not be place between quotes.

SIMPLE EXAMPLES
Example 1

```
#include<iostream.h>
int main()
{
        cout<<"Hello World!";
        return 0;
}
```

output: Hello world!

Example 2

```
#include<iostream.h>
int main()
{
        int k = 25;
        cout<<"The value in k is: "<<k;
        return 0;
}
```

output: The value in k is: 25

NOTE
The cout statement in the example above can be broken into two statements as follows.
cout<<"The value in k is:"
cout<<k;

The arrow notation << is often called insertion operator.
Hint: To remember which side the arrows should face, think of cout as the monitor so we are saying output should be taken "to the monitor" and hence the direction.

Example 3

```
#include<iostream.h>
int main()
{
int a = 20, b = 15;
cout<<a<<" + "<<b<<" = "<<(a + b);
return 0;
}
```

Output: 20 + 15 = 35

2.9 INPUT (Using cin)

cin is used for input i.e. to store values entered via the keyboard in a specified variable e.g.
1. cin>>k;
This value entered is stored in the variable k

2. cout <<"Enter the radius of the circle: ";
cin>>rad;
The user is prompted for the radius of and the value he / she enters is stored in a variable named rad.
3. cout<<"Enter a number: ";
cin>>a;
cout<<"Enter another number: ";
cin>>b;
The user is prompted for a number which is then stored in the variable a. The use is then prompted for another number which is then stored in the variable b.
The above statement can be combined as follows.
cout<<"Enter two number: ";
cin>>a>>b;
The user is prompted for two numbers. The first number entered is stored in the variable a and the second one is stored in the variable b.
You can have any number of variables for one cin e.g.
cin>>a>>b>>c>>d;
This statement tells the computer to wait for four values and then stores the first value in variable a, the second in variabe b, the third in variable c and the fourth in variable d.
During data entry the values should be separated by a space e.g. 2, 4, 12, 25

HINT
To remember which side the arrows (>>) should face, think of cin as the keyboard. So we are saying the value should come from the keyboard to the variable (and hence the direction) >> is called extraction operator.
Simple examples:

```
#include<iostream.h>
int main()
{
int a,b,c,sum;
double avg;
cout<<"\nEnter the first number: ";
cin>>a;
cout<<"\nEnter the second number: ";
cin>>b;
cout<<"\nEnter the third number: ";
cin>>c;
sum = a + b + c;
avg = sum / 3.0;
cout<<"\nsum = "<<sum;
cout<<"\nAverage = "<<avg;
return 0;
}
```

The above program can be re-written as: -

```
#include<iostream.h>
int main()
{
int a,b,c,sum;
double avg;
cout<<"Enter three numbers: ";
cin>>a>>b>>c;
sum = a + b + c;
avg = sum / 3.0;
cout<<"Sum = "<<sum<<"\nAverage = "<<avg;
return 0;
}
```

Example 2
Program to calculate the area and perimeter of a rectangle.

```
#include<iostream.h>
int main()
{
double length, width, area, perimeter;
cout<<"Enter the length and width of the rectangle: ";
cin>>length>>width;
area = length * width;
perimeter = 2 * (length + width);
cout<<"\nAre of the rectangle = "<<area;
cout<<"\nPerimeter = "<<perimeter;
return 0;
}
```

Example 3
Program to get the area and circumference of a circle.

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

```
int main()
{
double rad,circ,area;
cout<<"Enter the radius of the circle: ";
cin>>rad;
area = 22.0 / 7 * rad * rad;
circ = 22.0 / 7 * rad * 2;
cout<<"\nThe area of the circle = "<<area;
cout<<"\nCircumference = "<<circ;
return 0;
}
```

Approach 2

```
#include<iostream.h>
int main()
{
double rad, circ, area;
const double PI = 22.0 / 7;
cout<<"Enter the radius of the circle: ";
cin>>rad;
area = PI * rad * rad;
circ = PI * rad * 2;
cout<<"\nThe area of the circle = "<<area;
cout<<"\nCircumference = "<<circ;
return 0;
}
```

2.10 FORMATTING FOR NUMBERS WITH DECIMAL POINT

When you are outputting a value which has a fraction point (e.g. a number of type double or float) you might want to the value to be output correct to a given number of decimal places. To achieve that your program should contain some sort of instructions that tell the computer how to output the numbers. To output the value correct to two decimal places for example insert the following code segment in your program.

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

You may use any non-negative number in place of 2 to specify a different number of digits after the decimal point.

EXPLANATION

setf is an abbreviation for set flags. A flag is an instruction to do something in one of two ways. If a flag is given as an argument to setf, then the flag tells the computer to write output to that stream in some specific way. What it causes the stream to do depends on the flag.
In the first call to setf the flag **ios::fixed** causes the stream to output numbers of type double in what is called **fixed point notation** which is a fancy phrase for the way we normally write numbers.
If you wanted the numbers output in e – notation (e.g. 123e+02 which is the e-notation of 123), you use **ios::scientific i.e. (cout:setf::scientific); ios** – stands for input output stream.
In the second call to **setf ios::showpoint** tells the streams to always include a decimal point in floating numbers such as numbers of type double. If it is not set a number with all zeros after the decimal point might be output without the decimal point and the following zeros.
In the third line there is a call to a function called precision for cout which specifies the number of digits a precision after the decimal.
Example

```
#include<iostream.h>
int main()
{
double k=123.128672,p=42.46432;
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout<<"K = "<<k;
cout<<"\nP = "<<p;
return 0;
}
```

output:

```
K = 123.13
P = 42.46
```

Width Formatting Function
Consider the following example.

```
cout<<"Start here";
cout.width(4);
cout<<7;
```

output: Start here   7
The output has exactly three spaces between e and 7. The width function tells the stream how many spaces to use when giving an item as output. In this case, the item,

namely the number 7 occupies only one space and width said to use 4 spaces, so 3 of the spaces are blank. If the output requires more space then you specify as much additional space as is needed. The entire item is always output no matter what argument you give to width.
A call to width applies only to the next output. If you want to output twelve numbers using 4 spaces to output each number, then you must call width twelve times.

ADDITIONAL FORMATTING FLAGS FOR SELF
ios::showpos
If this flag is set, a plus sign is output before positive values.
N.B. Minus sign appears before negative numbers without setting any flags.

ios::right
If this flag is set and some field – width values is given with a call to the number function width, the next item output will be at the right end of the space specified by width. In other words, any extra blanks are placed before the item output. This flag is normally set by default.

ios::left
Functions in an opposite manner to ios::right i.e. the item is output on the left side of the space specified by width. In other words, any extra blanks are placed after the item.
Any flag may be unset. To unset a flag, use unsetf. E.g. cout.unsetf(ios::showpos);
This statement causes your program to stop including the plus sign on positive numbers.

2.11 MANIPULATIONS

Manipulations are stream number functions that are designed to do various things like set the width or precision of the output.
Manipulations are placed after the insertion operator <<, just as if the manipulator function call were an item to be output. Some examples of manipulations are setw and setprecision.

setw
Does exactly the same thing as the function width discussed earlier. This manipulation is called by writing if after the insertion operator << as if it were output.
Example.

```
cout<<"Start "<<setw(4)<<10<<setw(4)<<20<<setw(6)<<30;
output: Start 10    20      30
```

N.B. There are 2 spaces before the 10,two spaces before 20 and four spaces before 30.

setprecision
Does exactly the same thing as the member function precision discussed earlier. However, a call to set precision is written after the insertion operator <<

Example:

```
        cout.setf(ios::fixed);
        cout.setf(ios::showpoint);
        cout<<"Kshs. "<<setprecision(2)<<2500.0<<endl
                <<Kshs. "<<1200.5<<endl;
```

```
output: Kshs. 2500.00
        Kshs. 1200.50
```

Just like in the case of precision, when you set the number of digits after the decimal using setprecision, it stays in effect until you reset the it to some other number by another call to either setprecision or precision.
In order to use the manipulations setw and setprecision you must include the following directive in your program. #include<iomanip.h>

3 OPERATORS AND EXPRESSION

An operator specifies an expression to be performed that yields a value. An operand is an entity on which an operator acts. Operators are classified arithmetic, relational, logical and assignment operator. This can be further categorized into unary, binary or ternary operators depending on the number of operands they require.

3.1 ARITHMETIC OPERATORS

+ Plus
- Minus
* Asterisk for multiplication
/ Slash for division
% modulus operator for remainder (cannot be applied to float or double)

3.1.1 HIERARCHY OF OPERATIONS

Though the expressions are presented by a sequence of operators and variables, the C ++ compiler follows a hierarchy to execute them in order. The hierarchy of an arithmetic expression is given below.

1. * / % - Evaluated from left to right
2. + - Evaluated from left to right

The expressions within parenthesis are evaluated first following the above hierarchy. The expression in the inner most parenthesis are evaluated first.

EXAMPLES

1. x + y – z /* x and y are added first and then z is subtracted from the net result */
2. a + b / c /* b is divided by c and then the net result is added with a */
3. a + b * c /d % e ((8 * c) + 30 / (b * d))

3.1.2 Increment / Decrement operators

In some environments, situations may arise to increase or decrease a value by 1 continuously for sometime. C++ provides special characters ++ and – to do this.
This increment can be of two types:

i. First increment or decrement, then take the value for processing (prefix)
ii. Take the value for processing and the increment or decrement the variable (post fix)

3.1.3 Prefix Increment / Decrement

Examples:

1. K = 2; /* K is assigned with the value 2 */
   i = ++k; /* K is incremented by 1, then assigned to i therefore i = 3 and k = 3 */

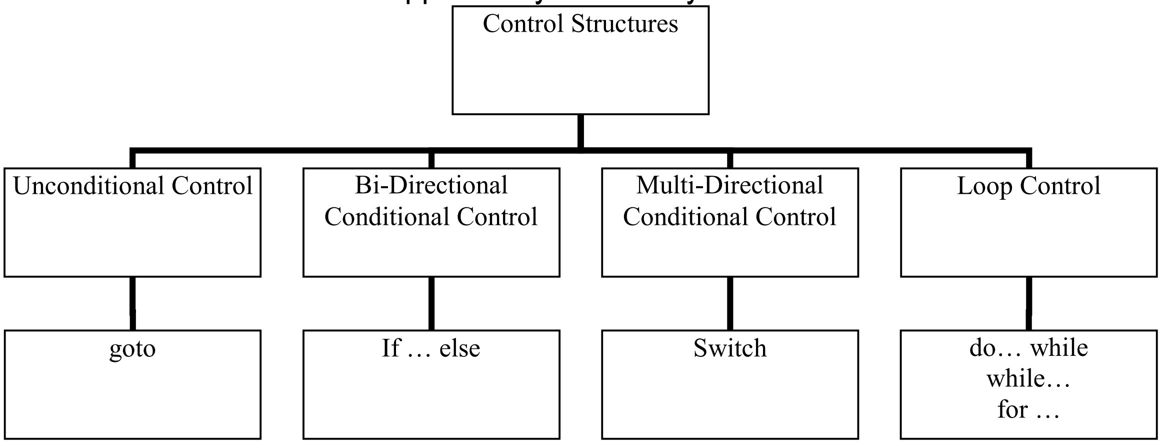
2. x = 25; /* x is assigned the value 25 */
   Y = -- x; /* x is decremented by 1 and then assigned to y.
   Therefore y = 24 and x = 24 */

3.1.4 Postfix increment / decrement

1. m = 20; /* m is assigned the value 20 */
   h = m++; /* h is assigned the value in m and then m is incremented by 1.
   Therefore h = 20 and m = 21 */
2. p = 30; /* p is assigned the value of 30 */
   Q = p--; /* q is assigned the value in p and then p is decremented by 1.
   Therefore q = 30 and p = 29 */

4 C++ CONTROL STRUCTURES

Usually, the statement in a C++ program are executed sequentially but in some environments, this sequential flow of execution might have to be changed and the control transferred to another part of the program. In such environments execution flow control structures are used.
The control structures define the way of flow in which the C++ statements should take place to achieve the required results.
For different structures are supported by C++. They are described below.



4.1 UNCONDITIONAL CONTROL (goto)

In some situations, the flow of control may be transferred to another part of the program without testing any condition. In such environments, goto is used. goto should be followed by a label name.
Syntax: goto identifier; /*where identifier is a label name */
Usually, goto makes a program unstructured. Too many goto statements will confuse the programmer and also make debugging very difficult.

Example 1

```
#include<iostream.h>
int main()
{
        int a, b;
        goto second;
        first:
        cout<<"\nEnter the first integer: ";
        cin>>a;
        goto ending;
        second:
        cout<<"\nEnter the second integer: ";
        cin>>b;
```

```
            goto first;
            ending:
            cout<<"\nThe numbers you entered were " << a << "and" << b;
        return 0;
        }
```

SAMPLE DIALOG

```
Enter the second integer: 12
Enter the first integer: 20
The numbers you entered were 20 and 12
```

4.2 BI-DIRECTIONAL CONDITION CONTROL

On a given condition, a block of statements can be executed using the bi-directional conditional control if … else. In this control, if a condition is tested and if it is true a single or a block of statements are executed.
The else can optionally be present along with if. Hence if … else can be presented in two ways. These are:
Syntax 1:

```
If (condition)
        Statement

If (condition)
{
        Statement 1
        Statement 2
        Statement last
}
```

Syntax 2

```
If (condition)
        Statement;
else
        statement;

if(condition)
{
        Statement1;
        Statement2;
        Statement3;
        Statement last;
}
Else
```

```
{
        Statement1;
        Statement2;
        Statement3;
        Statement last;
}
```

4.3 RELATIONAL / COMPARISON OPERATORS IN C++

== Equal to
!= Not equal to
< less than
> greater than
<= less than or equal to
>= greater than or equal to

4.4 LOGICAL / BOOLEAN OPERATORS IN C++

Logical operators, combine the results of one or more expressions and results in what is called a logical expression. After testing the condition, they return the logical status (true or false as the net result).

4.4.1 NOT (!)

It returns true if the expression is false and true otherwise. Syntax !(expression)
Example:

```
If (!(x<y))
        Statement;
```

The statement will only be executed if x is not less than y

4.4.2 AND (&&)

Returns true if both conditions being tested are true and false otherwise.
Syntax: (condition1 && condition2)
Example:

```
If ((x>=0 && (x<=100))
        Statements;
```

The statement will only be executed if x is greater than or equal to 0 and at the same time less than or equal to 100. i.e. x is between 0 and 100

4.4.3 OR (||)

Returns false if both conditions are false and true otherwise

Syntax: (Condition 1) || (condition 2)
Example:
If ((x<0) || (x>100))
The statement will be executed if either x is less than 0 or if x is greater than 100.

2. if ((a>10) || (b<700))
        Statements;
The statement will be executed if a is greater than 10 OR b is greater than 700 or if a is greater then 10 and b is less than 700.

NB. For or (||) only one condition has to be true (or both). The only time the overall result becomes false is only when both conditions are false.

Examples:
Example 1
A program that accepts 2 integers and checks which is bigger between them.

```
#include<iostream.h>
int main()
{
int a,b;
cout<<"\nEnter two integers: ";
cin>>a>>b;
if (a>b)
        cout<<a << "is bigger than " << b;
else
        cout<<b << "is bigger than "<< a;
return 0;
}
```

Example 2
A program that checks if some one is old enough to vote.

```
#include<iostream.h>
int main()
{
int age;
cout <<"\nEnter your age: ";
cin>>age;
if (age >=18)
        cout<< "You are old enough to vote: ";
else
        cout<< "You are too young to vote: ";
return 0;
}
```

Example 3:
A program that accepts three numbers and finds the biggest among them.
Approach 1

```
#include<iostream.h>
int main()
{
int a, b, c;
cout<<"Enter three integers: ";
cin>>a>>b>>c;
if(a > b && a > c)
        cout<< "The biggest number is "<< a;
else if (b > a && b > c)
        cout<<"The biggest number is " <<b;
else
        cout <<"The biggest number is"<<c;
return 0;
}
```

Approach 2

```
#include<iostream.h>
int main()
{
int a, b, c, big;
cout<<"Enter three integers: ";
cin>>a >> b >>c;
if (a > b && a > c)
big = a;
else if (b > a && b > c)
        big = a;
else
        big = c;
cout << "Between " << a << ","<<b <<" and " << c <<"the biggest is "<< big;
return 0;
}
```

Exercise

1. Write a program that accepts the marks a student gets in four subjects and then computes their average. The program should then assign the student an average grade based on the average marks. The grading system is given below.

Average Marks	Grade
80 – 100	A
70 – 80	B
60 – 70	C
50 – 60	D
0 – 50	E

2. Write a program that accepts an integer and checks whether it is even or odd.
3. Write a program that accepts an integer and checks whether it is divisible by 7.
4. A number is said to be evenly divisible by 9 if it is even and also divisible by 9. For example 18 is evenly divisible by 9 but 27 is not. Write a program that accepts an integer and checks whether or not it is evenly divisible by 9.

Example 4

In a certain organization, if an employee earns 20,000 or more, he is given a house allowance of 12% and transport allowance of 22% otherwise he is given a house allowance of 10% and transport allowance of 20%. A program that accepts basic salary and computers his / her net salary is shown below.

```
#include<iostream.h>
int main()
{
double basic, h_allow, t_allow, net_sal;
cout<<"\nEnter the employees basic salary: ";
cin>>basic;
if(basic >=20000)
{
        h_allow =12.0/100*basic;
        t_allow = 22.0/ 100*basic;
        net_sal = basic + h_allow + t_allow;
}
else
{
        h_allow =10.0/100*basic;
        t_allow = 22.0/100*basic;
        net_sal = basic + h_allow + t_allow;
}
cout<<"\nEmployee basic salary = "<<basic;
cout<<"\nHouse Allowance = "<<h_allow;
cout<<"\nTransport Allowance = "<<t_allow;
cout <<"\nNet salary = "<<net_sal;
return 0;
}
```

Note:

1. The last four cout statements can be contained into one as discussed earlier.
2. note the use of curly braces ({ and }) in the if…else statement.

4.5 MULTIDIRECTIONAL CONDITIONAL CONTROL

A multi-directional conditional control statement is one through which the control flow can be transferred in multi-directions. In c++ switch does this:

NB: You can use if…else
Syntax:

```
Switch(expression)
{
        case constant 1:
        block of statements;
        break;
        case constant 2:
        block of statements;
        break;
        …..
        default:
        block of statements;
}
```

Example 1

A program that accepts the points a student scored and determines the award using the system below:

Points	Award
4	Distinction
3	Credit
2	Pass
1	Fail

```
        #include<iostream.h>
        int main()
        {
        int points;
        cout<<"Enter the points the student score: ";
        cin>>points;
        switch(points)
        {
        case 4:
                cout<<"The student got a Distinction";
        break;
```

```
case 3:
        cout<<"The student got a Credit";
break;
case 2:
        cout<<"The student got a Pass";
break;
case 1:
        cout<<"The student got a Fail";
break;
default:
        cout<<"Invalid Points";
}
return 0;
}
```

The above program can be re-written using the if…else as follows:

```
#include<iostream.h>
int main()
{
int points;
cout<<"Enter student points: ";
cin>>points;
if (points==4)
        cout<<"The student got a Distinction";
else if (points==3)
        cout<<"The student got a Credit";
else if (points==2)
        cout<<"The student got a Pass";
else if (points==1)
        cout<<"The student got a Pass";
else
        cout<<"Invalid Points";
return 0;
}
```

Example 2
A program that accepts a character and checks whether it is a vowel.
Approach 1:

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

```
{
char lett;
cout<<"Enter a character: ";
cin>>lett;
switch(lett)
{
case 'a':
cout<<"a is a vowel";
break;
case 'e':
cout<<"e is a vowel";
break;
case 'i':
cout<<"i is a vowel";
break;
case 'o':
cout<<"o is a vowel";
break;
case 'u':
cout<<"e is a vowel";
break;
default:
cout<<lett<<" is not a vowel";
}
return 0;
}
```

Approach 2:

```
#include<iostream.h>
int main()
{
char lett;
cout<<"Enter a character: ";
cin>>lett;
switch(lett)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
```

```
cout<<lett <<" Is a vowel.";
break;
default:
cout<<lett<<" is not a vowel";
}
return 0;
}
```

4.5.1 RULES TO SWITCH

1. The expression value in switch [as in switch (expression)], must be an integer, hence the type can be int or char.
2. Case should always be followed by an integer constant, character constant or constant expression.
3. All cases should be distinct.
4. The block of statements under default is executed when none of the cases match the value of expression. Default can optionally be present. If it is not present and no case is matching, then no action takes place.
5. Cases and default can occur in any order.
6. The expression is evaluated first and the result is matched with each constant present in the cases.
7. Execution starts at the case which matches the value of the expression in switch.
8. The break statement causes an explicit exit from the switch statement. If it is not present, after executing the case which matches the value of expression. The following other cases are executed, the reason being that C++ treats the cases just as labels.

5 LOOP CONTROL STRUCTURES

A situation may arise, where you might need to repeat a statement or block of statements until a given condition s satisfied. Such repetitive statement or block is called loop control structure. C++ supports three types of loop control structures. They are:-
i. do…while
ii. while…
iii. for

5.1 do…while

syntax
```
do
{
        statement 1;
        statement 2;
        statement n;
}while (expression);
```

The block of statements is executed first. Then the expression is evaluated. The block of statements is evaluated repeatedly until the value of expression is false.

Example 1
A program that prints Hello World on the screen 10 times.
```
        #include<iostream.h>
        int main()
        {
        int k;
        k=1;
        do
        {
                cout<<"Hello World\n";
                k=k+1;
        }while(k<=10);
        return 0;
        }
```

Example 2
A program that prints the values between 1 and 20 on the screen using the do… while loop.
```
        #include<iostream.h>
        int main()
        {
                int p;
                p=1;
        do
        {
                cout<<"\nP= "<<p;
                p=p+1;
        }while(p<=20);
        return 0;
        }
```

Example 3
A program that gets the sum of all numbers divisible by 7 between 50 and 200 using do…while loop.

```
        #include<iostream.h>
        int main()
        {
                int h,sum=0;
                h=50;
        do
        {
                if(h%7==0)
                        sum=sum+h;
                h=h+1;
        }while(h<=200);
        cout<<"The sum of all numbers divisible by 7 between 50 and 200 is
        "<<sum;
        return 0;
        }
```

5.2 While

Syntax

```
while (expression)
{
        statement 1;
        statement 2;
        statement n;
}
```

In while, the expression is evaluated first. The block of statements are then evaluated repeatedly until the value of expression is false.

Example 1

A program that prints Hallo World to the screen 15 times using the while loop.

```
#include<iostream.h>
int main()
{
        int k;
        k=1;
while (k<=15)
{
        cout<<"Hallo World\n";
        k=k+1;
}
return 0;
}
```

Example 2

A program that prints all odd numbers between 1 and 30 on the screen using the while loop.

```
#include<iostream.h>
int main()
{
int p;
p=1;
while (p<=30)
{
        if (p%2!=0)
                cout<<"\nP = "<<p;
                p=p+1;
}
return 0;
}
```

Example 3

A program that gets the sum all the even numbers and the product of all odd numbers between 1 and 100 using the while loop.

```
#include<iostream.h>
int main()
{
int k;
double sum_even=0, prod_odd=1;
k=1;
while (k<=100)
{
        if (k%2==0)
                sum_even=sum_even + k;
        else
                prod_odd=prod_odd * k;
        k = k +1;
}
cout<<"Sum of even numbers between 1 and 100 = "<<sum_even;
cout<<"Product of odd numbers between 1 and 100 = "<<prod_odd;
return 0;
}
```

5.2.1 do vs while

1. do…while loop is executed atleast once. This is because the expression is evaluated after the first execution.
2. while loop executed only if the expression is true. This is because the expression is evaluated before the loop is executed.
3. do…while loop is always executed atleast once whereas while loop will only be executed if the expression is true.

A program to illustrate the difference between do and while.
1. Using do…while control statement.

```
#include<iostream.h>
int main()
{
int k=100;
do
{
        cout<<k;
        k=k+1;
}while (k<=100);
return 0;
}
```

Output 100

Explanation.
In the do…while loop, the value of k is output before the condition is tested. Therefore, the value 100 is output and then the value of k is increased by 1 to 101. 101 is not less than 100 and therefore control exits from the loop.

2. Some example using while control statement.

```
#include<iostream.h>
int main()
{
int k=100;
while (k<=100)
{
        cout<<k;
        k=k+1;
}
return 0;
}
```

Output: Nothing is output

Explanation:

In the while control structure the condition is tested first and in this case, the value of k = 100 which is not less than or equal to 10 and therefore the while loop is not executed.

Example 4

A properties that gets the sum of all numbers that are evenly divisible by of between 1 and 300 using while loop.

```
#include<iostream.h>
int main()
{
        int h, sum=0;
        h = 1;
while(h <= 300)
{
        if (h%2 == 0 && h % 9 == 0)
                sum=sum+h;
        h = h + 1;
}
cout<<"Sum of all numbers that are evenly divisible by 9 between 1 and
300 = "<<sum;
return 0;
}
```

5.3 for

for loop structure has three expressions. The first one is the initialization expression, the second one is the condition expression and the third is an expression that is executed for each iteration.

Syntax:

```
for(exp1; exp2;exp3)
block of statements;
```

exp1, exp2 and exp3 are expressions.

exp1 is an initialization statement.
exp2 is a conditional expression. It is evaluated first and the block of statements are repeatedly executed until the value of exp2 becomes false.
The above loop is equivalent to the following while loop statement.

```
exp1;
while(exp2)
{
        statement;
        exp3;
}
```

Example 1
A program that prints Hello World on the screen 10 times using the for loop.

```
        #include<iostream.h>
        int main()
        {
                int i;
        for (i=1; i<=10; i++)
        {
                cout<<"Hello World\n";
        }
        return 0;
        }
```

NOTE:
Instead of using i++ you can also use i=i+1 or i+=1. It is also possible to increase / decrease by a value other than 1. e.g. for (i=1; i<=100; i=i+2) where the value of i is increased by 2.

Example 2
A program that prints the first 20 positive integers using a for loop.

```
#include<iostream.h>
int main()
{
int i;
for (i=1; i<=20; i++)
        cout<<"\n i = "<<i;
return 0;
}
```

Example 3
A program that accepts an integer and computes its factorial using a for loop.
N.B. The factorial of a number n is defined as 1 x 2 x 3 x … x n.
For example, the factorial of 5 i.e. !5 = 1 x 2 x 3 x 4 x 5 = 120

```
#include<iostream.h>
int main()
{
        int i,n;
        long int fact=1;
cout<<"Enter a number to find factorial: ";
cin>>n;
for(i=1; i<=n; i++)
fact=fact * i;
cout<<"The factorial of  "<<n <<" is "<<fact;
return 0;
}
```

6 BREAK AND CONTINUE IN LOOP STRUCTURES

6.1 break

This statement causes an immediate exit form the innermost loop structure. This is used when an exit from the loop statement is required other than the testing at the top or bottom. It can also be used with switch statement.

Example

A programmer that prints the value of i until i is 6. When i is 6, control comes out of the loop and prints goodbye.

```
#include<iostream.h>
int main()
{
        int i;
for(i=1; i<=10; i++)
{
if (i==6)
{
        cout<<"\n I like this number !";
        break;
}
cout<<"\nThe value of i is "<<i;
}
cout<<"\nGoodbye";
return 0;
}
```

Output

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
I like this number !
Goodbye
```

Explanation

The for loop above can be executed 10 times i.e. from i=1 to i=10. The loop is only executed 6 times though. This is due to the presence of break.

6.2 continue

This statement causes the next iteration of the loop structure. It cannot be used with switch statement. When applied with do or while loops, the condition testing takes place. With for loop next iteration takes place.
This statement can only be used in a loop body.

Example

The following program prints the value of i until i is 6. When i is 6, because of the continue statement, the loop iteration continues until i becomes 10.

```
#include<iostream.h>
int main()
{
int i;
for (i=1; i<=10; i++)
{
        if(i==6)
        {
                cout<<"\n I like this number !";
                continue;
        }
        cout<<"\nThe value of i is "<<i;
}
cout<<"\n Goodbye";
return 0;
}
```

Output

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
I like this number !
The value of i is 7
The value of i is 8
The value of i is 9
The value of i is 10
```

Example 4

A program to print multiplication tables for 1 to 5 using nested for loops.

```
#include<iostream.h>
int main()
{
    int i,j;
for (i=1; i<=10; i++)
{
    for(j=1; j<=10; j++)
        cout<<endl<< i <<" *" << j <<" = " <<(i * j);
        cout<<endl;
}
return 0;
}
```

NB.
Instead of using cout<<endl; you can use cout<<"\n"; They do exactly the same thing.

7 FUNCTIONS

If a block of statements is to be executed repeatedly only in one place of the program, then the loop control structures are used.
In some situations a block of statements will have to be repeatedly executed in many parts of the program. In such environments, the block of statements entry has to be repeated and this consume a lot of time and computer memory. In C++, functions are used to solve such problems.
A function is a **sub-program** which is meant to do certain tasks. C++ is object oriented program. It is designed to do all tasks through functions.
There are two types of functions in C++

i. Built in functions / Library functions
ii. User defined functions

7.1 LIBRARY FUNCTIONS

Some operations like taking the square root of a number, sine of a number e.t.c. are frequently used by many programmers in their programs. Writing a program to find the square root of a number every time in each program is a unnecessary and tiresome job. Such operations are programmed and stored in the C++ library so that they can be called through any program. These functions are called **Library Functions or Built in functions.**
Some examples of library functions include sqrt, pow e.t.c
In C++ the call to mathematical functions can be made using the preprocessor directive #include<math.h>

7.2 USER DEFINED FUNCTIONS

Apart from the library functions that are built in C++, users can also define functions to do a task relevant to their programs. Such functions are called user defined functions. These functions are coded by the programmer.

Example.
A simple program that uses two functions greetings() and bye() to print different messages on the screen.

```
#include<iostream.h>
void greetings();
void bye();
int main()
{
        cout<<"Call to the function greetings \n";
        greetings();
        cout<<"Call to the function bye\n";
        bye();
        return 0;
}
void greetings()
{
        cout<<"How are you doing my friend?\n";
}

void bye()
{
        cout<<"I have to go. Have a nice day \n";
}
```

7.3 ADVANTAGES OF FUNCTIONS

1. The complexity of the entire task can be divided into simple sub-tasks and function sub-programs can be written for each sub-task.
2. The sub-programs (functions) are easier to write, understand and debug.
3. A function can be shared by other programs.
4. Amendments to a single function do not affect the rest of the program.
5. Functions make it easy for work to be divided among a group of programmers working on the same project.
6. In C++ a function can call itself again. This is called recursiveness.

7.4 ARGUMENTS / PARAMETERS

The parameters or arguments are used to provide communication between the calling and the called functions.

i. The **actual parameters** are the parameters of the calling function and **formal parameters** are parameters of the called function
ii. The number and type of the actual parameters should match with those of the formal parameters.
iii. The parameters are optional, if no parameters are present, then an opening and closing parenthesis should be present next to the function name.

Factorial (n,fact) – n & fact are parameters
Calc( ) – no parameters

iv. The statements under function should be present within curly braces
v. Parameters can be passed by value or by reference

7.5 RETURN STATEMENT

This is used in functions to return a value to the calling function. Every function sub-program can end with return.
There can be more than one return function and it is optional for any sub-program.
Syntax:
return (expression);
The value returned by the called function through return can be used or discarded, return need not be present.
Note:
Function calls can appear in expressions of a calling function – if only return statement is present in the called function, otherwise in the calling function it can appear as a statement.

7.6 DEFINITIONS

7.6.1 Function Call

A function call is an expression consisting of the function name followed by arguments enclosed in parenthesis. If there is more than one argument, the arguments are separated by commas.
Example

i. k = pow(a,2);
ii. greetings ( );

7.6.2 FUNCTION PROTOTYPE

The function prototype describes how the function is called. It tells you all you need to know to write a call to the function. It tells you the name of the function, how many arguments the function needs and what type of arguments should be.

7.6.3 FUNCTION DEFINITION

A function definition describes how the function computes the value it returns. If you think of a function as a small portion with your program, then the function definition is basically the code for that small program. A function definition consists of **a function header** followed by a **function body.**
A **function header** is written the same way as the function prototype except that the header does not have a semicolon at the end.
The **function body** follows the function header and completes the function definition. The function body consists of declarations and executable statements enclosed within a pair of curly braces.

Example 1

A program that computes the area of a rectangle using a function.

```
#include<iostream.h>
double rect_area(double width, double length); /* Function prototype */
int main()
{
double ar,w,h;
cout<<"Enter the width and length of the rectangle: ";
cin>>w>>h;
ar=rect_area(w,h); /*Function call */
cout<<"The area of the rectangle is "<<ar;
return 0;
}
double rect_area(double width, double length) /*Function header /
heading */
{
        double area;
        area = width * length;      // Function body & function definition
        return area;
}
```

In the above example, w and h are the actual parameters while width and length are the formal parameters.
When the function call is executed, the values of the actual parameters (w and h in this example) are plugged in the formal parameters (width and length) in this example). i.e. the values in the actual parameters are substituted in the formal parameters. This substitution process is known as call by value mechanism.

Things to note in the above substitution process

1. It is the values of the arguments that are plugged in the formal parameters. If the arguments are variables the values of the variables, not the variables themselves are plugged in.
2. The first argument is plugged in for the first formal parameter in the parameter list, the second argument is plugged in for the second formal parameter in the list and so forth.

Example 2

A program that computes both the sum and product of three numbers using functions.

```
#include<iostream.h>
double get_sum(double x, double y, double z);
double get_prod(double x, double y, double z);
int main()
{
        double a, b, c, total, prod;
cout<<"Enter three numbers: ";
cin>>a >>b >>c;
total = get_sum (a, b, c);
prod = get_prod (a, b, c);
cout<<a << " + " <<b <<" + "<<c <<" = " <<total;
cout<<endl <<a << " * " <<b <<" * "<<c <<" = " <<prod;
return 0;
}

double get_sum (double x, double y, double z)
{
        double sum;
        sum = x + y + z;
        return sum;
}

double get_prod (double x, double y, double z)
{
        double product;
        product = x * y * z;
        return product;
}
```

7.6.4 EXERCISE

1. Write a program that computes both the area and circumference of a circle using functions.
2. Write a program that has a function call **factorial** which accepts an integer from the main function and returns the factorial.
3. A program is required that accepts the students **average** mark and assigns a grade using the following grading system below.

Average Marks	**Grade**
80 – 100	A
70 – 79	B
60 – 69	C
50 – 59	D
0 – 49	E

i. Write a program with a function called **get_grade** which accepts the average mark and **returns** the grade to the main function which then outputs it.
ii. Write a program with a function called **grade** which accepts the average mark and **prints** the grade.

4. A program is required that accepts three integers and finds the smallest.
   i. Write a program with a function called **small** which accepts the three integers entered in the main function and returns the smallest. The main function should then output the three numbers entered together with the smallest.
   ii. Write a program with the function called **smallest** which accepts the three integers entered in the main function and then prints them together with the smallest.
5. Write a **function definition** for a function called odd which accepts an integer and returns 1 to the calling function if the integer is odd and 0 if it is even.

7.7 DEFINITIONS

7.7.1 Local Variables

These are variables whose values can only be accessed by the function they are declared in. They are declared with the body of a function definition.

7.7.2 Global Variables

These are variables whose value, can be accessed by all the functions in a program. They are declared outside the body of all functions and outside the main function of your program.

7.8 THE BLACK BOX ANALOGY

A programmer who uses a function in a program needs to know **what** the function does (such as calculate the square root, net salary e.t.c) but should not need to know how the function accomplishes its task. This is often referred to as treating a function like a **black box**.

Designing a function so that it can be used as a black box is sometimes called **information hiding** to emphasize that the programmer acts as if the body of the function were hidden from view.

Writing and using functions as if they were black boxes is also called **procedural abstraction**.

When applied to a function definition, the principle of **procedural abstraction** means that your function should be written so that it can be used like a **black box**. This means that he programmer who uses the function should **not** need to look at the function body of the function definition to see how the function works. The function prototype and the accompanying comment should be all the programmer needs to know in order to use the function.

To ensure that your function definitions have this important property, there are some rules that you need should adhere to.

1. The prototype comment should tell the programmer any and all conditions that are required of the arguments to the function and should describe the value that is returned by the function when called with these arguments.
Example see the function prototype below.
double new_balance(double bal, double rate);
/* Returns the balance in a bank account after posting simple interest. The parameter bal is the old balance. The formal parameter rate is the interest rate */

2. All variables used in the function body (with the exclusion of formal parameters) should be declared in the function body.

7.9 STUBS AND DRIVERS

Sometimes you might need to test a function outside the program for which it was intended. In such a case, you write a special program to do the testing. Programs like this are called **driver programs**. These driver programs are temporary tools and can be quite minimal.
It is sometimes impossible or inconvenient to test a function without using some other function that has not yet been written or has not yet been tested. In this case, you use a simplified version of the missing or untested function. These simplified functions are called stubs.

7.10 OVERLOADING FUNCTION NAMES

C++ allows you to give two or more definitions to the same function name, which means you can use and re-use names that have strong intuitive appeal across a variety of situations. For example you could have three functions called max. One that computes the largest of two numbers, another that computes the largest of three numbers and yet another that computes the largest of four numbers.
When you give two or more function definitions the same function name that is called **overloading** the function name.

Example.
Suppose you are writing a program that requires you to compute the average of two number you might use the following function definition.

```
double average(double n1, double n2)
{
        return ((n1 + n2) / 20 );
}
```

Now suppose your program also requires a function to compute the average of three numbers, you might define a new function called average2 as follows:

```
double average (double n1, double n2)
{
        return ((n1+n2)/2.0);
}
```

Now suppose your program also requires a function to compute the average of three numbers, you might define a new function called average2 as follows.

```
double average2 (doube n1, double n2, double n3)
{
        return((n1 + n2 + n3)/3.0);
}
```

This will work, but in many programming languages as you have no choice but to do something like this. Fortunately, C++ allows a more elegant solution. In C++, you can simply use the function name average for both functions. In C++, you can use the following function definition in place of the function definition average2.

```
double average (doube n1, double n2, double n3)
{
        return( (n1 + n2 + n3) / 3.0 );
}
```

So that the function has two definitions. This is an example of overloading. In this case, the overloading function name is average.
When the computer encounters a call to a function name that has two or more definitions, it checks the number of arguments and the types of arguments in the function call to know which function definition to use.

Whenever you give two or more definitions to the same function name, the various function definitions must have different specifications for their arguments; that is any two function definitions that have the same function name must use different numbers of formal parameters or use format parameters of different types (or both).
You cannot overload a function name by giving two definitions that differ only in the type of the value returned.
The use of the same function name to mean different things is called polymorphism. Overloading is an example of polymorphism.

Example 1
A program that computes the sum of two, three and four numbers using the concept of function overloading (polymorphism).

```
#include<iostream.h>
double add (double a, double b);
double add (double a, double b, double c);
double add (double a, double b, double c, double d);
int main()
{
        double a,b,c,d;
        cout<<"\nEnter four numbers: ";
        cin>>a>>b>>c>>d;
        cout << a <<" + " << b <<" = " <<add(a,b);
        cout <<endl << a <<" + " << b <<" + " <<c <<" = " <<add(a,b,c);
        cout <<endl << a <<" + " << b <<" + " <<c <<" + " << d <<" = " <<add(a,b,c,d);
return 0;
}
double add (double a, double b)
{
        double sum;
        sum = a + b;
        return sum;
}

double add (double a, double b, double c)
{
        double sum;
        sum = a + b + c;
        return sum;
}

double add (double a, double b, double c, double d)
{
        double sum;
        sum = a + b + c + d;
        return sum;
}
```

Example 2

A program that computes the area of a rectangle and the area of circle using the concepts of function overloading (polymorphism).

```
#include<iostream.h>
double area (double rad);
double area (double width, double length);
int main()
{
        double radius, w, l, circ_area, rect_area;
        cout<<"\nEnter the radius of the circle :";
        cin>>radius;
        cout<<"\nEnter the width and length of the rectangle: ";
        cin>>w >>h;
        circ_area = area (radius);
        rect_area = area (w, l);
        cout<<"\nThe area of the circle is : "<<circ_area;
        cout<<"\nThe area of the rectangle is: "<<rect_area;
        return 0;
}

double area (double rad)
{
        double a;
        a = 22.0 / 7 * rad * rad;
        return a;
}

double area (double width, double length)
{
        double a;
        a = width * length;
        return a;
}
```

7.10.1 Exercise

1. Assume you have the following details.

    Volume of a cuboid = length * breadth * height
    Volume of a sphere = $^4/_3$ π r^3
    Volume of a cylinder = π r^2 h (h – Height)
    Using the concept of polymorphism, write a program that demonstrates function overloading.

2. A program is required which will have function that can get the biggest between two numbers or the biggest among three numbers. Write the program and use the concept of function overloading.

7.11 RECURSIVE FUNCTIONS

The process of a function calling itself is called recursiveness. This is a special feature in C ++ and is supported by very few programming languages.

Example
A program to find the factorial of a number using the concept of recursiveness.

```
#include<iostream.h>
int fact (int a);
int main()
{
        int factorial;
int n;
cout<<"\nEnter a number to find factorial: ";
cin>>n;
factorial = fact (n);
cout<<"\nThe factorial of "<<n <<"is "<<factorial;
return 0;
}
int fact (int a)
{
        int main();
        if (a == 0)
                return 1;
        else
                m = a * fact (a – 1);
        return m;
}
```

Explanation
In the beginning, the accepted number n is passed by value to a. a is compared to 0 and if it is equal to 0, the function returns the result to the main program. Otherwise the same function is called with the parameter a – 1. This is repeated until a becomes 0.

Discussion of this concept with 3 as the accepted integer.
In the main function, 3 is accepted and stored in n. This value is passed to a in the function fact ( ) i.e. a is assigned with the same value as n in the function call.

Steps Followed after the function call
i. Evaluation of fact () when a = 3
In the function fact () a is compare with 0. in this case, a is not equal to 0 hence the statement after else is executed i.e. **m = 3 fact (3 – 1)**
Now the function fact is called again with the parameter 2.

ii. Evaluation of fact when a = 2

In the function fact (), a is compared with 0. In this case a is not equal to 0 hence the statement after else is executed i.e. **m = 2 * fact (2 – 1)**
Now the function fact () is called again with parameter 1.

iii. Evaluation of fact when a = 1
In the function fact (), a is compared with 0. In this case a is not equal to 0 hence the statement after else is executed i.e**. m = 2 * fact (1-1)**
Now the function fact is called again with the statement 0

iv. Evaluation of fact when a = 0
In the function fact () a is compared with 0. In this case a = 0 hence the function returns 1.
Now the returned value is substituted to the expression in (iii).
Hence m becomes: **m = 3 * 2 * 1 * 1**
The final result 6 is returned to the main program where fact () was called originally.

Example:
A program to find the sum of digits of an accepted integer using recursiveness concept.

```
#include<iostream.h>
int sum (int b);
int main()
{
        int n, ans;
cout<<"\nEnter an integer: ";
cin>>n;
ans = sum(n);
cout<<"\nThe sum of the digits of "<<n<< " is "<<ans;
return 0;
}

int sum(int b)
{
        int total;
if (b==0)
        return 0;
else
        total = b%10 +sum(b/10);
return total;
}
```

Exercise.

1. Write a program to print the Fibonacci series upto the nth term using the concept of **recursiveness** (N.B. The user should decide how many Fibonacci terms he wants.)

2. Write a program that computes the product of digits of an accepted integer using recursiveness concept.

7.12 CALL BY REFERENCE PARAMETERS

Normally when a function is called its arguments are **substituted** for the formal parameters in the function definition. These mechanism which is the mechanism we have used in all the previous examples is called call – by – value mechanism. The second mechanism that can be used is called **call by reference** mechanism.
In the **call by reference**, we are passing the variable itself as opposed to call by value where we pass the value in the variable and as a result, any modification / change that we do to that parameter in the function will have effect in the passed variable outside it.
The difference between **call by value** and **call by reference** mechanisms is that in the call by value mechanism any changes you make to the formal parameters in the function do not affect the values in the actual parameters, whereas in the call by reference mechanism any changes you make to the parameters in the function affect actual parameters.
To indicate that a parameter is a call by reference parameter attach the ampersand sign (&) to the end of the type name in the formal parameters list both in the function prototype and the header of the function definition.

Example.

```
void get_input(double& f_variable)
{
        cout<<"Enter the temperature in fahreinheit: ";
        cin>>f_variable;
}
```

In a program that contains the function definition, the following function will set the variable f_temperature equal to the value read from the keyboard.
get_input(f_temperature);

A simple example to illustrate the difference between call by value and call by reference mechanism.

Example 1
Call by value

```
#include<iostream.h>
void my_function (int x, int y);
int main()
```

```
        {
                int a=5,b=10;
                cout<<"Before the function call :\n";
                cout<<"a = "<<a<<"\nb = "<<b;
                my_function(a,b);
                cout<<"\nAfter the function call: \n";
                cout<<"a = "<<a<<"\nb = "<<b;
                return 0;
        }

        void my_function (int x, int y)
        {
                x = x *10;
                y = y * 20;
                cout<<"\nInside the function :\n";
                cout<<"\nx = "<<x<<"\ny = "<<y;
        }
```

Output

```
Before the function call:
a = 5
b= 10
Inside the function:
X = 50
Y = 200
After the function call:
a = 5
b = 10
```

N.B. Even though the values of x and y (the formal parameters) have been changed, the value of a and b (the actual parameters) are not affected.

Example 2
Call by reference (same program)

```
        #include<iostream.h>
        void my_function(int& x, int&y);
        int main()
        {
                int a=5,b=10;
        cout<<"Before the function call:\n";
        cout<<"a = "<<a<<"\nb = "<<b;
        my_function(a,b);
        cout<<"\nAfter the function call: \n";
        cout<<"a = "<<a<<"\nb= "<<b;
        return 0;
        }

        void my_function(int& x, int& y)
        {
        x = x * 10;
        y = y * 20;
        cout<<"\nInside the function:\n";
        cout<<"x = "<<x<<"\ny = "<<y;
        }
```

Output

```
Before function call
a = 5
b = 10
Inside the function:
x = 50
y = 200
After the function call:
a = 50
b = 200
```

N.B. in this case, when the value of x is changed inside the function, the value of a also changed. When you change the value of y, the value of y also changes.

Example 3.
A program that accepts two integers and passes them by reference to a function which then interchanges the values in the two integers.

```
        #include<iostream.h>
        void swap (int& x, int&y);
        int main()
        {
        int a,b;
        cout<<"nEnter two integers :";
        cin>>a>>b;
        cout<<"a = "<<a<<"\n b = "<<b;
        swap(a,b);
        cout<<"\nAfter swap: \n a = "<<a<<"\n b = "<<b;
        return 0;
        }
```

```
void swap (int& x, int& y)
{
int temp;
temp = x;
x = y;
y = temp;
}
```

8 CLASSES

A class is a data type that has both data members and member functions. It can also be described as a data type whose variables are objects.
An object is a variable that has functions as well as data associated with it. Objects can also be defined as instances of a class.
A **member function** is a function that is associated with an object.
A class specifies both code and data. When you define a class, you declare the data it contains and the code that operates on that data. While very simple classes might contain only code or only data, most real world classes contain both.
Data is contained in instance variables defined by the class and code is contained in functions. The code and data that constitutes a class are called members of the class.
The general form of a class:
A class is by use of the keyword class
Syntax:

```
class class_name
{
permission_label_1;
        members;
permission_label_2;
        members;
}object_name;
```

class_name is a name for the class (user defined type) and optional field **object_name** is one or several, valid object identifiers. The body of the declaration can contain numbers, that can either be data or function declarations and **optionally permission labels** that can be any of these three keywords:- **private, protected or public.**
They make reference to the permission which the following members acquire.
Private members of a class are accessible only from other members of its class and from its friend classes. All member variables that are listed after the keyword private are referred to as private member variables, which means they cannot be directly accessed in the program except within the definition of a member function. You can also have **private member functions.**

Public members of a class are accessible from anywhere where the class is visible. This means that a public member can be used in the main body of your program or in the definition of any function even a member function.
Protected members are accessible to members of the same class, friend classes and members of its derived classes.
Taking all this into consideration, the above syntax can be refined to look as follows:-

```
class class_name
```

```
{
private:
        private member variables and functions
public:
        public member variables and functions
} object_list;
```

N.B. If we declare members of a class before including any permission label for the members they are considered private since it is the default permission that the members of a class declared within the class keyword acquire.

```
class declarations example

class rectangle
{
private:
        int x, y;
public:
        void set_values(int a, int b);
        int area();
}rect;
```

This is the declaration of a class called rectangle with private member variables x and y and public member functions set_values() and area(). rect has been declared as an object of the type rectangle.
The above class can also be declared as follows.

```
class rectangle
{
private:
        int x, y;
public:
        void set_values(int a, int b);
        int area();
};
```

If later you want to declare an object of type rectangle you use the line of code below.
rectangle rect1, rect2;
In the code above, rect1 and rect2 have been declared as objects of type rectangle.

8.1 MEMBER FUNCTION DEFINITION

A member function is defined the same way as any other function except that the class_name and the scope resolution operator (::) are given in the function heading.
The operator :: which is called the scope resolution operator is used in the function heading of a member function to indicate which class a member function is a member of. This is because you can have two or more classes with member functions with the same name for example, if you had two classes circ and rect both with member functions called area, the definition for the member function area in circ would have the header circ::area() and the one for the class rect would be rect::area()
The class name that precedes the scope resolution operator is called type qualifier.

Syntax:

```
returned_type class_name::function_name(parameter list)
{
        function body statements;
}
```

example

```
void DayOfYear::output()
{
        cout<<"\nMonth = "<<month<<",day = "<<day<<endl;
}
```

Example 1

A simple program that has a class called DayOfyear

```
#include<iostream.h>
class DayOfYear
{
public:
        int month,day;
};
int main()
{
DayOfYear today,birthday;
cout<<"\nEner today's date:\n";
cout<<"\nEnter month as a number: ";
cin>>today.month;
cout<<"\nEnter the day of the month: ";
cin>>today.day;
cout<<"\nEnter your birthday:\n";
cout<<"\nEnter month as a number: ";
cin>>birthday.month;
```

```
        cout<<"\nEnter the day of the month: ";
        cin>>birthday.day;
        cout<<"\nToday's date is:\n";
        cout<<"Day = "<<today.day<<",Month = "<<today.month;
        cout<<"\nYour Birthday is :\n";
        cout<<"Day = "<<birthday.day<<",month = "<<birthday.month;
        if (today.day==birthday.day && today.month==birthday.month)
                cout<<"\nHappy Birthday!";
        else
                cout<<"\nHappy unbirthday!";
        return 0;
        }
```

N.B. The above example is not standard. This is because the main part of the program has direct access to the member variables of the class. Ideally, the main part of the program should not have direct access to the member variables of a class and therefore these variables should normally be private. This helps to make the main part of the program immune to any changes made in the member functions of the class. When you make member variables private, they can only be accessed via member functions of the class. See example below.

Example 2
The example above re-written in standard manner.

```
#include<iostream.h>
class DayOfYear
{
private:
        int month,day;
public:
        void input(), output();
        int get_month(), get_day();
};

int main()
{
DayOfYear today,birthday;
cout<<"Enter today's date:\n";
today.input();
cout<<"Enter your birthday:\n";
birthday.input();
cout<<"\n Today's date is:\n";
today.output();
cout<<"\n Your birthday is: \n";
birthday.output();
if(today.get_day()==birthday.get_day() &&
today.get_month()==birthday.get_month())
        cout<<"\nHappy Birthday";
else
        cout<<"\nHappy unbirthday!";
return 0;
}
void DayOfYear::input()
{
cout<<"Enter the month as a number: ";
cin>>month;
cout<<"\nEnter the day of the month: ";
cin>>day;
}

void DayOfYear::output()
{
cout<<"Day = "<<day<<" ,Month = "<<month;
}

int DayOfYear::get_month()
{
        return month;
}

int DayOfYear::get_day()
{
        return day;
}
```

8.2 THE DOT OPERATOR

The dot operator links the name of an object with the argument of a member (member function or member variable)

Syntax:

```
        Object.member;
```

Examples

1.) today.day – Accessing the instance variable day for the object today.
2.) birthday.month – accessing the instance variable month for the object birthday.
3.) today.input()

4.) birthday.output()
NB. You don't use the dot operator in the member function because they have direct access to the members of their class.

8.3 ACCESSOR FUNCTIONS

As discussed earlier, when you make member variables private you cannot be able to access them directly form the main part of the program. If you wanted to access the values of such member variables, e.g. in the above example where we wanted to test if today is somebody's birthday, you have to define member functions to provide that access. Such member functions are called accessor functions.
Accessor functions are member functions that give you access to the values of the private member variables. In the above examples, get_day and get_month are accessor functions.

8.4 ENCAPSULATION

It is a programming mechanism that binds together code and the data it manipulates and that keeps both safe from outside interference and misuse. For example, when you combine a number of items, such as variables and functions into a single package, such as an object of same class, this is called encapsulation.

EXAMPLE 3
A program that uses a class named my_circle to accept the radius of a circle and compute both the area and circumference.

```
#include<iostream.h>
class my_circle
{
private:
        double radius, circum, area;
public:
        void calculate(),input(),output();
};

int main()
{
my_circle circle;
circle.input();
circle.calculate();
circle.output();
return 0;
}
```

```
void my_circle::input()
{
        cout<<"Enter the radius of the circle: ";
        cin>>radius;
}

void my_circle::output()
{
        cout<<"\nThe area ="<<area<<"\n Circumference = "<<circum;
}

void my_circle::calculate()
{
        circum=22.0 / 7 *radius *2;
        area = 22.0 / 7 * radius * radius;
}
```

8.5 CONSTRUCTORS (For initialization)

A constructor is a member function that is automatically called when an object of that class is declared and its purpose is to initialize the values of some or all member variables and to do any other sort of initialization that may be needed.
A constructor is defined in the same manner as any other member function, except for two points.

1. A constructor must have same name as the class. For example if the class is named BankAccount, then any constructor for this class must be named BankAccount.
2. A constructor definition cannot return a value. Moreover, no type not even void can be given at the start of the function prototype or in the function header.

For example, suppose we wanted to add a constructor for initializing the radius in the previous example (example 3), the class definition should be as follows.

```
class my_circle
{
public:
        my_circle(double rad);
        void calculate(), input(), output();
private:
        double radius, circum, area;
};
```

Notice that the constructor is named my_circle, which is the name of the class. Also notice that the prototype for the constructor my_circle does not start with void or with any other type name. Finally, notice that the constructor is placed in the public section of class definition. Normally if you were make all your constructors private member thus you would not be able to declare, any objects of that class type which would make the class completely useless.

Write a redefined class my_circle, two objects of type my_cirlcle can be declared and initialized as follows.

```
        My_circle circle1(7), circle2(21);
```

Assuming that the definition of the constructor performs the initializing action we promised, the above declaration will declare the object circle1 and set the value of circle1.radius to 7. Similarly, the value of circle2.radius is set to 21.
The result is conceptually equivalent to the following (although you cannot write it this way in C++)

```
my_cirlce circle1, circle2; // problems but fixable
circle1.my_circle(7); // very illegal
circle2.my_cirlce(21); // very illegal
```

the first line can be made acceptable by including a **default constructor.**
The next two lines (the calls to the constructor my_circle) are, illegal. A constructor cannot be called in the same way as an ordinary member function is called (using a dot operator). If you want to call the constructor again to reinitialize the object to new values you use the assignment operator (=) rather than the dot operator.
For example, in the above example if you have already used the value for radius for circle1 and you want to reinitialize it to 28 at a different part of the program, you use circle1=my_circle(28);
The definition for a constructor is given in the same way as any other member function for example, if you revised the definition of the class my_circle by adding a constructor just described, you need also to add the following definition of the constructor.

```
        My_circle::my_circle (double rad)
        {
                radius = rad;
        }
```

Since the constructor function and the class have the same name, my_circle appears twice in the function heading. The my_circle before the scope resolution operator :: is the name of the class and the my_circle after the scope resolution operator is the name of the constructor function. Also notice that no return type is specified in the heading of the constructor definition, not even type void. Aside from these points, a constructor is defined in the same way as an ordinary member function.

N.B. You can overload a constructor name (like my_circle::my_circle) just as you can overload any function name. In fact constructors are usually overloaded, so that objects can be initialized in more than one way.

Example 4

A program that uses a class named Rectangle to compute the area and perimeter of a rectangle. The class has a constructor that is used to initialize the values of width and length. The program also illustrates how to reinitialize these values after the computations have been performed.

```
#include<iostream.h>
class Rectangle
{
public:
        Rectangle(double a, double b);
        void input(), compute(), output();
private:
        double width, length, area, perim;
};
int main()
{
Rectangle rect(10,5);
rect.compute();
cout<<"Rectangle 1:\n";
rect.output();
cout<<"\nRectangle 2:\n";
rect=Rectangle(20,10);
rect.compute();
rect.output();
cout<<"\n\nRectangle 3:\n";
rect.input();
rect.compute();
rect.output();
return 0;
}
Rectangle::Rectangle (double a, double b)
{
        width = a;
        length = b;
}

void Rectangle::compute()
{
        area = length *width;
        perim = 2 * (length + width);
}

void Rectangle::output()
{
        cout<<"The area of the rectangle = "<<area<<"\nPerimeter = "
            <<perim;
}

void Rectangle::input()
{
        cout<<"Enter the width and length of the rectangle: ";
        cin>>width>>length;
}
```

N.B. The above program seeks to illustrate all the things covered so far. Your program doesn't have to include all the functionalities illustrated.

8.5.1 DEFAULT CONSTRUCTOR

Using constructors is an all or nothing situation. If you give atleast one constructor for a class, then every time you declare an object for that type, C++ will look for an appropriate constructor definition to use.
For example, suppose you define a class as follows:

```
class SampleClass
{
public:
        SampleClass (int parameter1, double parameter)
// constructor that requires 2 parameters
void do_stuff();
private:
        int data1;
        double data2;
};
```

You should recognize the following as a legal way to declare an object of type SampleClass and call the constructor for that class.
SampleClass my_object(7, 7.7);
However, you'll be surprised to learn that the following is illegal.
SampleClass your_object;
Since SampleClass has a constructor, the compiler interprets the above declaration as including a call to a constructor with no arguments, but there is no definition for a

constructor with zero arguments. You must either add two arguments to the declaration of your object or else you must add a constructor definition with no arguments.
A constructor with no arguments is called the default constructor since it applies in the default case where you declare an object without specifying any arguments. Since it is likely that you will sometimes want to declare an object without giving constructor arguments, it's a good habit to always include a default constructor.
If you redefine the class SampleClass as follows, then the above declaration of your_object would be legal.

```
class SampleClass
{
public:
        SampleClass(int parameter1, double parameter2);
        SampleClass();
void do_stuff();
private:
        int data1;
        double data2;
};
```

The default constructor can initialize the member variables to any default values you choose e.g. 0 for example if you wanted the default constructor above to initialize the values of the member variables to zero, its definition would be:-

```
SampleClass:SampleClass
{
        // do nothing
}
```

PITFALL
If a constructor for a class called SampleClass has two formal parameters, you declare the object and give the arguments to the constructor as follows:-
SampleClass my_object(10, 12.5)
To call a constructor with no arguments, you would naturally think that you would declare the object as follows.
SampleClass your_object(); // This will cause problems.
After all, when you call a function that has no arguments, you include a pair of empty parentheses. However, this is wrong for a constructor. Moreover, it may not produce an error message, since it does have an unintended meaning. The compiler may think that the above is a prototype for a function called your_object that takes no argument and returns a value of type SampleClass.
You don't include parenthesis when you declare an object and want C++ to use the constructor with no arguments. The correct way to declare your_object using the constructor with no arguments is:- SampleCalss your_object;

However, if you call a constructor explicitly in an assignment statement (to reinitialize the values) you do use the parenthesis e.g.
my_object = SampleClass();

Example 5
A program that, illustrates the use of overloading constructors including a default constructor.

```
#include<iostream.h>
class BankAccount
{
public:
        BankAccount (int shillings, int cents, double rate);
        BankAccount (int shillings, double rate);
        BankAccount (); // Default constructor
void output();
private:
        double balance, interest_rate;
};
int main()
{
BankAccount account1(1000,50,10), account2(500,5), account3;
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout<<"Account 1 is initialized as follows";
account1.output();
cout<<"\nAccount 2 is initialized as follows:";
account2.output();
cout<<"\nAccount 3 is initialized as follows: ";
account3.output();
return 0;
}
BankAccount::BankAccount (int shillings, int cents, double rate)
{
        balance = shillings + 0.01 *cents;
        interest_rate = rate;
}
BankAccount::BankAccount (int shillings, double rate)
{
        balance = shillings;
        interest_rate = rate;
}

BankAccount::BankAccount()
{
        balance=0;
        interest_rate=0;
}

void BankAccount::output()
{
        cout<<"\nAccount balance = Kshs ."<<balance<<"\nRate =
"<<interest_rate<<"%";
}
```

NOTE

You can write the function definitions for all member functions (methods) including constructors in the class declarations / definition. In such a case, you don't repeat the classname in the function headings. You also don't include the scope resolution operator. See the example below.

Example 6

```
#include<iostream.h>
class Numbers
{
public:
Numbers (double x, double y)
{
        num1 = x;
        num2 = y;
}
Numbers()
{
//Default constructor
}
void output()
{
cout<<"Enter two numbers: ";
cin>>num1, num2;
}
void calculate()
{
        results = num1+num2;
}

void output()
{
        cout<<num1<<" + "<<num2 " = "<<results;
}
private:
        double num1, num2, results;
};

int main()
{
Numbers set1(25,6), set2;
cout<<"Data entry for set 2:\n";
set2.input();
set1.calculate();
set2.calculate();
cout<<"\nOutput for set1: \n";
set1.output();
cout<<"\nOutput for set2: \n";
set2.output();
return 0;
}
```

NB. The same approach can be used for all the examples much earlier.

8.6 DESTRUCTORS

A **destructor** is a member function of a class that is called automatically when the object of the class goes out of scope. If an object is a local variable for a function, then the destructor is called automatically as the last action before the function call ends. For global objects, the destructor is called when the program ends. Destructors are mainly used to reallocate memory that had previously been allocated to the object.
A destructor (just like a constructor) always has the same name as the class if is a member of, but the destructor has a tilde symbol (~) at the beginning of its name. like a constructor, a destructor has no type for the value returned, not even the type void.
A destructor has no parameter. Thus a class can only have one destructor, you cannot overload the destructor of a class. Otherwise a destructor is defined just like any other member function.

Example

If you wanted to add a destructor to the BankAccount example met earlier (Example 5) you would add the following in the public part of the class definition.
~BankAccount();
You would then add a definition for the destructor e.g.

```
	BankAccount::~BankAccount()
	{
		cout<<"\n Destructing…";
	}
```

This destructor simply displays a message, but in real life programs the destructor would be used to release memory, close open files e.t.c.

8.7 ABSTRACT DATA TYPES (ADIs)

A data type is called an **abstract data type (ADI**) if the programmer who uses the type need not have access to the details of how the values and operations are implemented. The predefined types such as int are abstract data types (ADIs). This is because you don't know how implementation such as + and * are implemented for type int. even if you know you would not use this implementation in your program. Programmer defined types such as class types are not automatically ADIs. In order to define a class so that it is an abstract data type, you need to separate the specification of how the type is used by the programmer for the details of how the type is **implemented.** The separation should be so complete that you can change the **implementation** of the class and any program that uses the class ADI should not need any additional changes. One way to ensure this separation is to:
1. Make all the member variables private members of the class.
Make each of the basic operations that the programmer needs a public member function of the class and fully specifying how to use each such public member function.
Make any helping functions private member functions.

The **interface** of an ADI tells how to use the ADI in your program. When you define an ADI in a C++ class, the interface consists of the public member functions of the class along with the interface of the ADI should be all you need in order to use the ADI in your program.
The **implementation** of the ADI tells you how this interface is realized, as a C++ code. The implementation of the ADI consists of the private members of a class and the definition of both public and private member functions. Although you need the implementation in order to run a program that uses the ADI, you need not know everything about the implementation in order to write the rest of a program that uses the ADI.

8.8 FRIEND FUNCTIONS

A friend function is an **ordinary** function that has access to the private members of objects of that class. You make a function a friend of a class by listing the function prototype in the definition of the class and placing the keyword **friend in front** of the function prototype. The prototype may be placed in either the private section of the public section, but it will be a public function in either case, so it is clearer to list in the public section.

N.B. A friend function is not a member function. A friend function is defined and called the same way as an ordinary function. You don't use the dot operator in a call to a friend function and you don't use the type qualifier in the definition of a friend function.
Syntax: of a class definition with friend functions.

```
class class_name
{
public:
friend prototype_for_friend_function1
friend prototype_for_friend function2
member_function_prototypes;
private:
        private_member_declarations_and_prototypes
};
```

Example 7

```
#include<iostream.h>
class DayOfYear
{
public:
        void input(), output();
        friend void check(DayOfYear date1, DayOfYear date2);
private:
        int day, month;
};

int main()
{
DayOfYear today, b_day;
cout<<"Enter today's date: \n";
today.input();
cout<<"\nEnter your birthday:\n";
b_day.input();
cout<<"\nToday is: ";
today.output();
cout<<"\nYour birthday is: ";
b_day.output();
check(today,b_day);
return 0;
}
void DayOfYear::input()
{
        cout<<"Enter the day and the month (as integers): ";
        cin>>day>>month;
}

void DayOfYear::output()
{
        cout<<"\nDay = "<<day<<",Months = "<<month;
}

void check(DayOfYear date1, DayOfYear date2)
{
        if (date1.day == date2.day && date1.month==date2.month)
                cout<<"\nHappy Birthday!";
        else
                cout<<"\nHappy unbirthday!!!";
}
```

NOTE
A friend function (just like all other ordinary functions) can accept parameters or any data or none at all and can return values of any data type or none at all.
Member functions and friend functions serve a very similar role. In-fact sometimes it is not clear whether you should make a particular function a friend of your class or a member of the class because either would perform the same task in the same way.
A simple rule to help you decide between member functions and non member functions is the following:-
1. Use a member function if the task being performed by the function includes only one object.
Use a non member function if the task being performed involves more than one object (like in example 7)

8.9 INHERITANCE

Inheritance is the defining of a new class based on an already existing class so that the new class has all the members of the class it is inheriting from and it also adds in own unique elements.
A base class is a class that is inherited by another class. A derived class is a class that inherits from another class. A derived class inherits al l the members defined in the base class and adds its own unique elements.
C++ implements inheritance by allowing one class to incorporate another class into its declarations. This is done by specifying the base class when a derived class is declared.
Syntax:

```
Class derived_class : access base_class
{
        // body of derived class
}
```

where derived_class is the name of the derived class, base_class is the name of the class being inherited. Here access is optional. However, if present, it must be public, private or protected. If the access specifier is not used, then it is private by default.
If the access specifier is public we will have public inheritance.
Public inheritance is a form of inheritance where all public members of the base class become pubic members of the derived class.

Example

```
class B: public A
{

}
```

If the access specifier is private we have private inheritance. Private inheritance is a form of inheritance where all pubic members of the base class become private members the derived class.
Example

```
class B: private A
{

}
```

If the access is protected all public and protected members of the base class become protected members of the derived class.

NOTE:
If a class is inherited as public, private or protected the private members of the base class remain private to that class and are not accessible by members of the derived class.
When a class is inherited as public, protected members of the base class become protected members of the derived class. When a base class is inherited as private, protected members of the base class become private members of the derived class.

Example:

```
class TwoDShape
{
protected:
        double width, height;
public:
        void showdim(), setdim(double a, double b);
};

class Triangle: public TwoDShape
{
public:
        char style[20];
        double area();
        void showstyle();
};
```

In the above example, there are two classes, TwoDShape (which is the base class) and Triangle (which is inherits TwoDShape). In the example, Triangle inherits all the members of TwoDShape and also adds its own unique elements.
If we had declared width and height in TwoDShape as private variables they wouldn't accessible from any member function of Triangle (revisit the definations). We therefore wouldn't be able to calculate the area of the triangle. If we made them public variables, they would be accessible by all parts of the program including main which is not desirable. That is why we used protected. Protected variables work like private variables but the advantage is that they can also be accessed by member functions of the derived class. (Revisit the definitions)
N.B. A base class is a completely independent stand alone class and therefore it is possible to declare objects of either the base class of the derived according to your requirements. In the above example, you can declare objects of type TwoDShape or Triangle according to your requirements.

Example 8

```
#include<iostream.h>
class Employees
{
protected:
        double basic, t_allow, h_allow, net_sal;
public:
        void get_input(), get_output(), output_net();
};
class Manager: public Employees
{
private:
        double ent_allow, r_allow;
public:
        void m_input(), m_output();
};

int main()
{
Employees emp;
Manager mng;
cout<<"Employee details: \n";
emp.get_input();
cout<<"manager's details: \n";
mng.m_input();
cout<<"Employee Analysis: \n";
emp.get_output();
emp.output_net();
cout<<"\n\nManager's Analysis: \n";
mng.m_output();
mng.output_net();
return 0;
}

void Employees::get_input()
{
cout<<"Enter the basic salary: ";
cin>>basic;
cout<<"\nEnter the transport allowance: ";
cin>>t_allow;
cout<<"\nEnter the house allowance: ";
cin>>h_allow;
net_sal = basic + t_allow + h_allow;
}

void Employees::get_output()
{
cout<<"\nBasic salary: "<<basic;
cout<<"\nTransport Allowance: "<<t_allow;
cout<<"\nHouse Allowance: "<<h_allow;
}

void Manager::m_input()
{
        get_input();
cout<<"\nEnter the entertainment allowance: ";
cin>>ent_allow;
cout<<"\nTne the responsibility allowance: ";
cin>>r_allow;
net_sal = net_sal + ent_allow + r_allow;
}

void Manger::m_output()
{
get_output();
cout<<"\nEntertainment allowance: "<<ent_allow;
cout<<"\nResposibility allowance: "<<r_allow;
}

void Employees::output_net()
{
        cout<<"\nNet Salary: "<<net_sal;
}
```

NOTE:
I have called get_input() in Manager::m_input() and get_output() in Manager::m_output to reduce repetition. It is perfectly legal though to repeat the statements get_input() in Manager::m_input instead of making the function call. The same applies to get_output.

Example 3

```
#include<iostream.h>
class Vehicle
{
```

```
private:
        char make[20], c_of_m[20];
        int y_of_m;
public:
        void v_input(), v_output();
};

class Matatu: public vehicle
{
private:
        char route[15];
        int speed_limit, passengers;
public:
        void m_input(), m_output();
};

int main()
{
Vehicle personal;
Matatu mat;
cout<<"\nData entry for personal car: ";
personal.v_input();
cout<<"\n\nData entery for your matatu:\n";
mat.m_input();
cout<<"\n\nAnalysis for the personal car:\n";
personal.v_output();
cout<<"\n\nAnalysis for your matatu:\n";
mat.m_output();
return 0;
}
void Vehicle::v_input()
{
        cout<<"Enter the make of your Vehicle: ";
        cin>>make;
cout<<"\nEnter the country of manufacture: ";
cin>>c_of_m;
cout<<"Enter its year of manufacture: ";
cin>>y_of_m;
}

void Vehicle::v_output()
```

```
{
        cout<<"\nMake : "<<make;
        cout<<"\nCountry of Manufacture: "<<c_of_m;
        cout<<"\nYear of Manufacture: "<<y_of_m;
}

void Matatu::m_input()
{
v_input();
cout<<"\nEnter the matatu's route: ";
cin>>route;
cout<<"\nEnter its speed limit: ";
cin>>speed_limit;
cout<<"Enter the max no of passengers: ";
cin>>passengers;
}

void Matatu::m_output()
{
        v_output();
        cout<<"\nRoute: "<<route;
        cout<<"\nSpeed Limit: "<<speed_limit<<" km/h";
        cout<<\nNo of passengers: "<<passengers;
}
```

8.10 CONSTRUCTORS AND INHERITANCE

When only the derived class defines a constructor, the process is straight forward. The constructor in the derived class can initialize either the member variables of the derived class or both the member variables of the base class and of the derived class.

Example:
If in example 8, employees didn't have a constructor and we wanted to define a constructor for manager which is the derived class, we would include the following in the class definition for Manager.

```
Manager (double bsc, double t_a, double h_a, double e_a, double r_a)
{
basic = bsc;
t_allow = t_a;
h_allow = h_a;
ent_allow = e_a;
r_allow = r_a;
```

```
        net-sal = bsc + t_a + h_a + e_a + r_a;
        }
```

Calling base class constructor.
When a base class has a constructor, the derived class must explicitly call it to initialize the base class portion of the object. A derived class can call a constructor defined by its base class by using an expanded form of the derived class constructor declaration. The general form of this expanded declaration is shown here.

```
derived_constructor (arg list): base_constructor (arg list)
{
        body of derived constructor;
}
```

example: if in example 8, Employees had a constructor, we would have included the following in its definition

```
        Employees(double bsc, double t_a, double h_a)
        {
                basic = bsc;
                t_allow = t_a;
                h_allow = h_a;
                net_sal = basic + t_allow + h_allow;
        }
```

If we wanted to include a constructor the derived class (Manager) then will have to make an explicit call to the constructor defined above in the function header for the constructor for Manager. That constructor would look as shown below: -

```
        Manager(double bsc, double t_a, double h_a, double e_a, double r_a):
        Employees (bsc, t_a, h_a);
        {
                ent_allow = e_a;
                r_allow = r_a;
                net_sal = net_sal + ent_allow + r_allow;
        }
```

N.B.
In this case the constructor for Manager is just initializing the members of its class and then passing the other values to the constructor for the basic class.

8.10.1 Exercise.

1. Write a program that makes use of a class called Biz to accept the buying price of an item, transport and selling price and then compute the profit of loss. The class should have three member functions, input(), calculate() and output().
2. Write a program that makes use of class called Volume to compute the area and volume of a sphere. The program should have two constructors, one to initialize the radius to a given value and a default constructor that initializes the radius to 0. Declare two objects to type Volume with the first one initializing the radius to 10 and the second one making use of the default constructor. The user should enter the radius for the second object. In both cases, your program should calculate and display both the area and volume of the spheres.
3. Write a program that computes the compound interest earned by an amount of money deposited in a bank account. Your program should make use of a class called Bank_Int with all member variables being private. The input for your program should be deposit, interest rate and the number of years the money is being saved. The output should be the values entered above, the interest earned and the bank balance. Define two constructors one that initializes the deposit, interest rate and the number of years to given values and a default constructor that initializes these values to 0. Declare two objects in your class, one that makes use of each of the constructors.

9 ARRAYS

An array is a data structure that enables us to declare a group of variables of the same data type under a common name.

9.1 Array declaration

Arrays should be declared as any basic type with an integer value given within square brackets. By declaring an array, the specified number of locations are fixed in the memory.
Naming an array follow the same rules as those of a variable.
There is no limit for an array dimension in C++. The limit can only be the computers main memory.

Syntax:
type var[n]; //one dimensional array
type – variable type e.g. int, double e.t.c
var – variable name
n – positive constant or constant expression. This represents the maximum number of elements that the array can hold.
Example:
int mark[10];
In the above example, an array called mark has been declared and it can hold a maximum of 10 integers. These integers can be referred to as mark[0], mark[1], mark[2]… mark[9]. Reference to array elements always starts from 0. In this particular array (i.e. mark[10]) there is no element called mark[10].

Example 1
A program that accepts 10 numbers, stores them in an array and then displays the array elements.

```
#include<iostream.h>
int main()
{
        int i, mark[10];
for (i=0;i<10;i++)
{
cout<<"\nEnter number "<<(i+1)<<" : ";
cin>>mark[i];
}
cout<<"\nThe numbers you entered were: \n";
for(i=0;i<10;i++)
cout<<mark[i]<<" ";
return 0;
}
```

Example 2
A program that accepts 10 numbers and finds out the biggest, smallest, sum and average.

```
#include<iostream.h>
int main()
{
int i;
double num[10], big, small, sum = 0, average;
for(i=0; i<10; i++)
{
cout<<"\nEnter number "<<(i+1)<<" : ";
cin>>num[i];
}
big=small = num[0];
for(i=0;i<10;i++)
{
        sum = sum +num[i];
        if (num[i]>big)
                big = num[i];
        else if(num[i]<small)
                small = num[i];
}
average = sum / 10;
cout<<"\nThe biggest number is: "<<big;
cout<<"\nThe smallest number is: "<<small;
cout<<"\nThe sum is "<<sum<<" and the average is "<<average;
return 0;
}
```

9.2 ARRAY INITIALIZATION

Arrays can be initialized at the time of declaration as other variables. In the following example reg_no is declared to be an integer array of 5 elements and all are initialized with numbers.

Example.

```
int reg_no[5]={10, 20, 30, 40, 50};
```

9.3 RULES TO INITIALIZE AN ARRAY

1. Arrays can only be initialized with constants
2. The whole array can be referenced by the array name
3. Fewer initializers than the specified size are allowed. Example int reg_no[5] = {10, 20}; In this case, reg_no[0] is initialized with 10 and reg_no[1] is initialized with 20. All the rest are automatically initialized with 0.
4. Too many initializers then the specified size gives erroneous results.
5. The middle elements cannot be initialized alone. It should always be preceded by the initialization for the previous elements. For example, in the above example, if reg_no[2] alone has to be initialized there is no way to do it in the declaration. The value of reg_no[0] and reg_no[1] also should be present.
6. If all the elements are to be initialized with a specific number, the repetition of the data cannot be avoided. For example if, reg_no[20] has to be initialized with 1 at the time of declaration, 1 has to be typed 20 times.
7. Arrays can be initialized without mentioning the number of elements. It is derived automatically by the compiler with respect to the number of elements inside the braces.

Example

Int num[ ] = {1,2,3,4,5,}; /* Array size is 5 */

Example 3

A program that initializes an array with 10 marks and finds out the number of passed and failed candidates (if the mark is >=50, the student has passed.)

```
#include<iostream.h>
int main()
{
int i, pass, fail;
double mark[10] = {89, 78, 43, 28, 49, 56, 88, 79, 19};
pass = fail = 0;
for(i=0; i<10; i++)
{
if  (mark[i]>=50)
            pass = pass + 1;
else
            fail = fail + 1;
}
cout<<"The number of candidates who passed = "<<pass;
cout<<"\n The number of candidates who failed  = "<<fail;
return 0;
}
```

9.4 ARRAY OF OBJECTS

If you want to declare a few objects of a certain class you can declare them individually.

Example

```
Circle circ1, circ2, circ3;
```

In the above example, three objects (circ1, circ2, circ3) of the class Circle have been declared.

In some instances, we might want to declare many objects say 10, 20 or even 100 of the same class. In other instances we might not know exactly how many objects the user will require at one particular time. For example, if it is a class holiday student details, the user might want to work with 20 students at one time and 30 students at another. In the above two scenarios, it would not be very practical to declare the objects individually, we instead declare an array of objects.

Example

```
Circle circ[100];
```

The above is a declaration of an array called circ which can hold a maximum of 100 objects of the class Circle.

Example

A program that computes both the area and perimeter of any number of rectangles using classes. The user is given a chance to specify how many rectangles he want to work with.

```
#include<iostream.h>
class Rectangle
{
public:
        void get_input(), compute(), output();
private:
        double width, length, area, perim;
};

int main()
{
Rectangle rect[50];
int i, n;
cout<<"How many rectangles do you want to work with: ";
cin>>n;
for (i=0; i<n; i++)
{
cout<<"Data entry for rectangle "<<(i+1)<<":\n";
rect[i].get_input();
rect[i].compute();
}

cout<<"\n\nOutput";
for(i=0; i<n; i++)
{
        cout<<"\n\nRectangle "<<(i+1) <<":\n";
        rect[i].output();
}
return 0;
}

void Rectangle::get_input()
{
        cout<<"Enter the length and width of the rectangle: ";
        cin>>width>>length;
}

void Rectangle::compute()
{
        area = length * width;
        perim = 2 * (width + length);
}

void Rectangle::output()
{
        cout<<"\nWidth = "<<width<<"\nLength = "<<length;
        cout<<"\nArea = "<<area<<"\nPerimeter = "<<perim;
}
```

9.5 PASSING ARRAYS TO FUNCTIONS

There are three ways of declaring a parameter that receives an array from a calling function.
It can be declared as an array of the same type and size as the array in the calling function.

Example

```
#include<iostream.h>
void display(int num[10])
int main()
{
        int numbers[10],I;
for(i=0; i<10; i++)
{
        cout<<"Enter number: "<<(i+1)<<" : ";
        cin>>numbers[i];
}
display(numbers);
return 0;
}
void display(int num[10])
{
int k;
cout<<"\n\nThe values in the array are: \n";
for (k=0; k<10; k++)
        cout<<num[k]<<" ";
}
```

NOTE
In the function call, we are only using the array name. i.e. display (numbers); not display (numbers[10]); this is because the second would be interpreted as passing the 11th item in the array numbers (which actually doesn't exist.)

2. Declare an array parameter to specify an unsized array.
In the above example, you can change the function prototype to void display(int num[ ]); and the function header to void display (num[ ]) leaving everything else intact.

3. Declare the parameter as a pointer. This is the method most commonly used in professionally written C++ program.
In the above example, you can change the function prototype to void display(int *num); and the function header to void display(int *num) leaving everything else intact.

N.B.
When you pass an array to a function, it is passed by reference and therefore any changes made in the function to the values will affect the values in the array in the calling function.

9.6 MULTI – DIMENSIONAL ARRAY

An array can have more than one dimension and it can be represented by more than one subscript. The following is an example of a two dimensional array. An array of more than one dimension is considered to be an array of arrays.
int employee[10][50];

employee[0] is considered to be an array of five elements referenced as employee[0][0], employee[0][1] …. employee[0][4]
The same applies to employee[1] … employee[9]

Example 1
A program that accepts values and stores them in a 2 x 2 matrix (a two dimensional array) and then prints out the matrix.

```
#include<iostream.h>
int main()
{
int arr[2][3];
int i, j;
for(i=0; i<2; i++)
        for (j=0; j<3; j++)
{
cout<<"\n Enter element (" <<(i+1)<<","<<(j+1)<<"): ";
cin>>arr[i][j];
        }
cout<<"\n\nThe matrix you entered was: \n";
for (i=0; i<2; i++)
{
for (j=0; j<3; j++)
        cout<<arr[i][j]<<" ";
        cout<<"\n";
}
return 0;
}
```

Example 2
A program to add two matrixs

```
#include<iostream.h>
int main()
{
float a[10][10], b[10], c[10][10];
int i, j, m, n;
cout<<"\n Enter the order of the two matrices m and n: ";
cin>>m>>n;
for (i=0; i<m; i++)
for(j=0; j<n;j++)
{
        cout<<"\nEnter A ("<<(i+1)","<<(j+1)<<"): ";
        cin>>a[i][j];
```

```
	}

	for (i=0; i<m; i++)
	for(j=0; j<n;j++)
	{
		cout<<"\nEnter B ("<<(i+1)","<<(j+1)<<"): ";
		cin>>b[i][j];
	}

	for (i=0; i<m; i++)
	for(j=0; j<n;j++)
		c[i][j] = a[i][j] + b[i][j];
		cout<<"\n\nMatrix A and B are \n\n";

	for(i=0; i<m; i++)
	{
		for(j=0; j<m; j++)
			cout<<a[i][j];
		cout<<"\t";
		for (j=0; j<m; j++)
			cout<<b[i][j];
		cout<<"\n";
	}
	cout<<"\nThe resultant when these two matrices are added is: \n\n";
	for (i=0; i<m; i++)
	{
		for(j=0; j<m; j++)
			cout<<c[i][j]<<" ";
		cout<<"\n";
	}
	return 0;
	}
```

9.6.1 INITIALIZING A TWO DIMENSIONAL ARRAY

Example

```
	int num[2][4] = { {1,2,3,4}, {5,6,7,8} };
```

NB

We initialize row by row. In the above example num has 2 rows and 4 columns. We have initialized the first row with 1,2,3 and 4 and the second row with 5,6, 7 and 8.

Example

A program that initializes a two dimensional array with values and then prints out the array elements together with their sum.

```
#include<iostream.h>
int main()
{
int arr[2][3] = { {10, 20, 30}, {40, 50, 60} };
int i, j, sum = 0;
cout<<'The array elements are: \n";
for (i=0; i<2; i++0
{
        for (j=0; j<3;j++)
        {
                sum = sum + arr[i][j];
                cout<<arr[i][j]<<" ";
        }
        cout<<"\n";
}
cout<<"\n\nTheir sum is: "<<sum;
return 0;
}
```

9.7 STRINGS AND CHARACTER ARRAYS

An array of characters can be declared and hence a chain of characters called a string can be stored in that string. In the example below, name is character array of 20 elements.

```
char name[20];
```

C++ places the end of string character (\0) as the last character of the string.

9.7.1 STRING LIBRARY FUNCTIONS

To manipulate strings, you can use the following inbuilt functions.

i. strlen (s1) – returns the length of a string s1
ii. strcmp(s1, s2) – returns a value
  < 0 when s1 <s2
  = 0 when s1 = s2
  >0 when s1> s2
iii. strcpy (s1, s2) – copies s2 to s1
iv. strcat (s1, s2) concatenates s2 at the end of s1

Example

A program that accepts a name and then displays it.

```
#include<iostream.h>
int main()
{
        char name[20];
        cout<<"\n Enter your name: ";
        cin>>name;
        cout<<"\nThe name you entered was "<<name;
return 0;
}
```

9.7.2 INITIALIZING A CHARACTER ARRAY

```
1. char city[10]={'N', 'a', 'i', 'r', 'o', 'b', 'i', '\0'};
or
2. char city[10]="Nairobi"
```

10 STRUCTURES

A **structure** can be defined as a group of variables of different data types organized together under a single name.
Structures are inherited from C language and are declared using the keyword **struct**. A structure is syntactically similar to a **class** and both create a class type. In C language, a structure can contain only data members but this limitation does not apply to C++, the structure is essentially just an alternative way to specify a class. In-fact in C++ the only difference between a class and a structure is that by default all members are **public** in a **structure** and **private** in a **class**. In all other aspects, structures and classes are equivalents. As a result, in all the previous examples, we can simply replace the keyword class with the keyword struct and the programs would still work properly.
To avoid confusion, C++ programmers normally use a class to define the form of an object that contains member functions and struct in its more traditional role to create objects that contain only data members.

Syntax for declaring structures (The traditional way)

```
struct structure_tag
{
        type members;
};
```

The structure_tag specifies the name of the structure and its optional. You can also have variables optionally declared between the closing curly brace (}) and the semicolon.
The structure members can be referenced only with the structure variable as a qualifier separated by the dot operator. The members are referenced as follows:-
name.member

Declaration example

```
struct
{
        int reg_no;
        char name[20];
        float total,average;
}x,y;
```

Here x and y are declared to be two structure type variables. The parts of the structure are reg_no, name[20], total and average and are called structure members or structure elements.

Example 2

```
struct stud_rec
```

```
{
        int adm_no;
        char name[20];
        float height;
};
stud_rec year1, year2, year3;
```

In the above example, a structure called stud_rec is defined. Three variables i.e. year1, year2 and year 3 are then declared to be variables of this structure type. For year1 you can access the member variables as year1.adm_no, year1.name, year1.height. The same applies to year2 and year3.

Example 1
An example that, illustrates the use of structures. A program that accepts your admission number, name, age and height and prints them out.

```
#include<iostream.h>
struct Student
{
        char adm_no[15],name[20];
        int age;
        float height;
};
int main()
{
        Student stud1;
        cout<<"Data entry for the student:\n"
                <<"-----------------------------------------------------------\n";
        cout<<"Enter the admission number of the student: ";
        cin>>stud1.adm_no;
        cout<<"Enter his/her name: ";
        cin>>stud1.name;
        cout<<"Enter his/her age: ";
        cin>>stud1.age;
        cout<<"Enter his/her height: ";
        cin>>stud1.height;

        cout<<"Analysis for the student:"
                <<"\n-------------------------------------"
                <<"\nAdmission Number: "<<stud1.adm_no
                <<"\nName: "<<stud1.name
                <<"\nAge: "<<stud1.age
                <<"\nHeight: "<<stud1.height;
```

```
          cout<<"\n\n";
          return 0;
}
```

Example 2

```
#include<iostream.h>
#include<stdlib.h>
struct Student
{
          char adm_no[15],name[20];
          int age;
          float height;
};
int main()
{
          Student stud1,stud2;
          cout<<"Data entry for student 1:\n"
                    <<"--------------------------------------------------------------
\n";
          cout<<"Enter the admission number of the student: ";
          cin>>stud1.adm_no;
          cout<<"Enter his/her name: ";
          cin>>stud1.name;
          cout<<"Enter his/her age: ";
          cin>>stud1.age;
          cout<<"Enter his/her height: ";
          cin>>stud1.height;

          cout<<"\n\nData entry for student 2:\n"
                    <<"-------------------------------------------\n";
          cout<<"Enter the admission number of the student: ";
          cin>>stud2.adm_no;
          cout<<"Enter his/her name: ";
          cin>>stud2.name;
          cout<<"Enter his/her age: ";
          cin>>stud2.age;
          cout<<"Enter his/her height: ";
          cin>>stud2.height;

          system("cls");
```

```
        cout<<"Analysis for student 1:"
              <<"\n-------------------------------------"
              <<"\nAdmission Number: "<<stud1.adm_no
              <<"\nName: "<<stud1.name
              <<"\nAge: "<<stud1.age
              <<"\nHeight: "<<stud1.height;

        cout<<"\n\nAnalysis for student 2:"
              <<"\n-------------------------------------"
              <<"\nAdmission Number: "<<stud2.adm_no
              <<"\nName: "<<stud2.name
              <<"\nAge: "<<stud2.age
              <<"\nHeight: "<<stud2.height;
        cout<<"\n\n";
        return 0;
}
```

10.1 STRUCTURES AND ARRAYS

Structure arrays can be defined, so that each array element can be of structure data type.

Example

```
struct stud_rec
{
        int adm_no, age;
        char name[20];
        float height;
}stud[50];
```

Here, stud is an array of 50 elements, each of which have the structure type stud_rec. such kind of array are called structure arrays.

Example 2

A program that accepts the admission number, name, age and height of a given number of students.

```
#include<iostream.h>
struct Student
{
        char adm_no[15],name[20];
        int age;
        float height;
};
int main()
{
        Student stud[50];
        int i, number;

        cout<<"How many students do you want to work with: ";
        cin>>number;
        for (i = 0; i < number; i++)
        {
                cout<<"Data entry for the student "<<(i + 1)<<":\n"
                        <<"---------------------------------------------------\n";
                cout<<"Enter the admission number of the student: ";
                cin>>stud[i].adm_no;
                cout<<"Enter his/her name: ";
                cin>>stud[i].name;
                cout<<"Enter his/her age: ";
                cin>>stud[i].age;
                cout<<"Enter his/her height: ";
                cin>>stud[i].height;
        }

        for (i = 0; i < number; i++)
        {

                cout<<"Analysis for student "<<(i + 1)<<": "
                        <<"\n-------------------------------------"
                        <<"\nAdmission Number: "<<stud[i].adm_no
                        <<"\nName: "<<stud[i].name
                        <<"\nAge: "<<stud[i].age
                        <<"\nHeight: "<<stud[i].height;
        }
        cout<<"\n\n";
        return 0;
}
```