CATHOLIC UNIVERSITY OF RWANDA

P.O. Box 49 BUTARE / HUYE - RWANDA
TEL: (00250) 0252 530 893 FAX: (00250) 0252 530 627
E-mail: administration@cur.ac.rw
Web: www.cur.ac.rw

FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

MODULE OF ADVANCEDEDPROGRAMMING USING C++ 'SEMESTER II'

LEVEL TWO

CREDITS: 10, HOURS: 50

ACADEMIC YEAR 2018-2019

LECTURER: NSENGIYUMVA Jean Marie Vianney

Email : nsengiyumva@gmail.com

PRE-REQUISITE OR CO-REQUISITE MODULES/COMPONENTS:

- **Computer Programming**

LEARNING OUTCOMES

Brief description of aims and content

Module consists of C++ Programming Language

- Introduces the fundamental concepts programming from an object-oriented perspective. Through the study of object design, this course also introduces the basics of human computer interfaces, graphics, and the social implications of computing, along with significant coverage of software engineering
- Introduces the advanced programming concepts

8. Learning outcomes

Having successfully completed the module, students should be able to:

1. Design, implement, test, and debug simple programs in C++ programming language.
2. Describe how the class mechanism supports encapsulation and information hiding.
3. Design, implement, and test the implementation of “is-a” relationships among objects using a class hierarchy and inheritance.
4. Compare and contrast the notions of overloading and overriding methods in an object-oriented language.
5. GUI programming features.
6. Programming for network and database servers
7. Programming for security enabled transactions.
8. Robust programming concepts.

LEARNING AND TEACHING STRATEGIES

Lectures

Practical works

Group & Individual work

Assignments

Practical works

Contents

Brief description of aims and content ... 2

8. Learning outcomes ... 2

CHAP 1: BASICS OF C++ ... 5

1.1 Introduction ... 5

1.2Evolution of Programming Languages ... 5

1.3Characteristics Of C++ ... 6

1.4Historical Development of C++ ... 7

1.5Application of C++ ... 8

1.6 Getting Started with C++ ... 9

CHAP 2: VARIABLES AND CONSTANTS ... 13

2.1 CONSTANTS, VARIABLES AND KEYWORDS ... 13

2.2. VARIABLES IN C++ ... 15

2.3 DEFINED CONSTANTS ... 20

2.4 INPUT/OUTPUT (I/O) ... 20

CHAP 3: ARRAYS ... 24

3.1 DEFINITION: ... 24

3.2.ONE-DIMENSIONAL ARRAY ... 25

3.3. Multidimensional Array ... 30

CHAP 4: THE USE OF CONDITIONS ... 34

4.1. THE IF STATEMENT ... 34

4.2THE IF-ELSE STATEMENT ... 37

4.3 CASE CONTROL STRUCTURES ... 45

CHAP 5: LOOP CONTROL STRUCTURES ... 51

CHAP 6: THE USE OF FUNCTIONS ... 59

6.1 DEFINITION ... 59

6.2. OVERLOADING FUNCTIONS: ... 76

CHAP 7: FRIENDSHIP AND INHERITANCE ... 80

7.1. FRIEND FUNCTIONS ... 80

7.2.Polymorphism ... 90

Variables and Memory ... 93

CHAP 1: BASICS OF C++

1.1 Introduction

Computers are able to perform many different tasks, from simple mathematical operations to sophisticated animated simulations.

Computer do not create these tasks by themselves, these are performed by following a series of predefined instructions that conform to what we call a program.

A computer does not have enough creativity to make tasks which it has not been programmed for, so it can only follow the instructions of programs which it has been programmed to run.

Those in charge of generating programs so that the computers may perform new tasks are known as **programmers or coders**, who for that purpose use a **programming language.**

1.2Evolution of Programming Languages

A programming language is a set of instructions that order computers on what to do.

When choosing a programming language to make a project, many different considerations can be taken. First, one must decide what is known as the *level* of the programming language.

The level determines how near to the hardware the programming language is.

In the lower level languages, instructions are written thinking directly on interfacing with hardware,
In "high level” a more abstract or conceptual code is written. High level code is more portable, that means it can work in more different machines with a smaller number of modifications, whereas a low level language is limited by the peculiarities of the hardware which it was written for. A higher or lower level of programming is to be chosen for a specific project depending on the type of program that is being developed.

Example: when a hardware driver program is developed for an operating system obviously a very low level is used for programming.
While when big applications are developed usually a higher level is chosen, or a combination of critic parts written in low level languages and others in higher ones.
The C++ language is in a middle position, since it can interact directly with the hardware almost with no limitations, and can as well work like one of the most powerful high level languages.

1.3Characteristics Of C++

Portability

You can practically compile the same C++ code in almost any type of computer and operating system without making any changes. C++ is the most used and ported programming languages in the **world.**

Brevity

Code written in C++ is very short in comparison with other languages, since the use of special characters is preferred to key words, saving some effort to the programmer.

Object-oriented programming

An application's body in C++ can be made up of several source code files that are compiled separately and then linked together.

Saving time since it is not necessary to recompile the complete application when making a single change but only the file that contains it.
In addition, this characteristic allows to link C++ code with code produced in other languages, such as Assembler or C.

C Compatibility

C++ is backwards compatible with the C language. Any code written in C can easily be included in a C++ program without making any

Speed

The resulting code from a C++ compilation is very efficient, due indeed to its duality as high-level and low-level language and to the reduced size of the language itself.

1.4Historical Development of C++

By 1960 many computer languages had come into existence, each was meant for a specific purpose. Example COBOL was used for Commercial Applications, FORTRAN for Engineering and Scientific Applications.

With this many languages in place programmers wanted to come up with one language which could be used to handle all possible applications. An international committee was set up and it came out with a language called ALGOL 60.
ALGOL 60 turned out to be too abstract and too general. A new language called Combined Programming Language (CPL) was developed at Cambridge to counter the ALGOLS 60 problem but it also turned to be too big, with so many features, which made it hard to learn, an to implement.
Martin Richards developed basic combined Programming Language at Cambridge (BCPL) University to overcome the problem of CPL, but it turned to be less powerful and too specific.

Around the same time Ken Thompson, wrote a language called B at AT & T's Bell Labs, but the B language also turned to be so specific.Dennis Ritchie inherited the features of B and CPL, added some of his own featured and developed the C Language. C++ Devolved from c. **Bjarne Stroustrup** took c and extended the feature needed to facilitate object oriented programming. He created C++ at the AT&T laboratories.

C++ is an object-oriented programming language. C++ is an extension of C with a major addition of the class construct feature .since a class is a major addition to the original C language. Bjarne Stroustrup Initially called the new language "**C with classes**"

However ,later 1n 1983,the name was changed to C++. The ideal of C++comes from the C increment operator ++, thereby suggesting that C++ is augmented (incremented) version of C.

C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

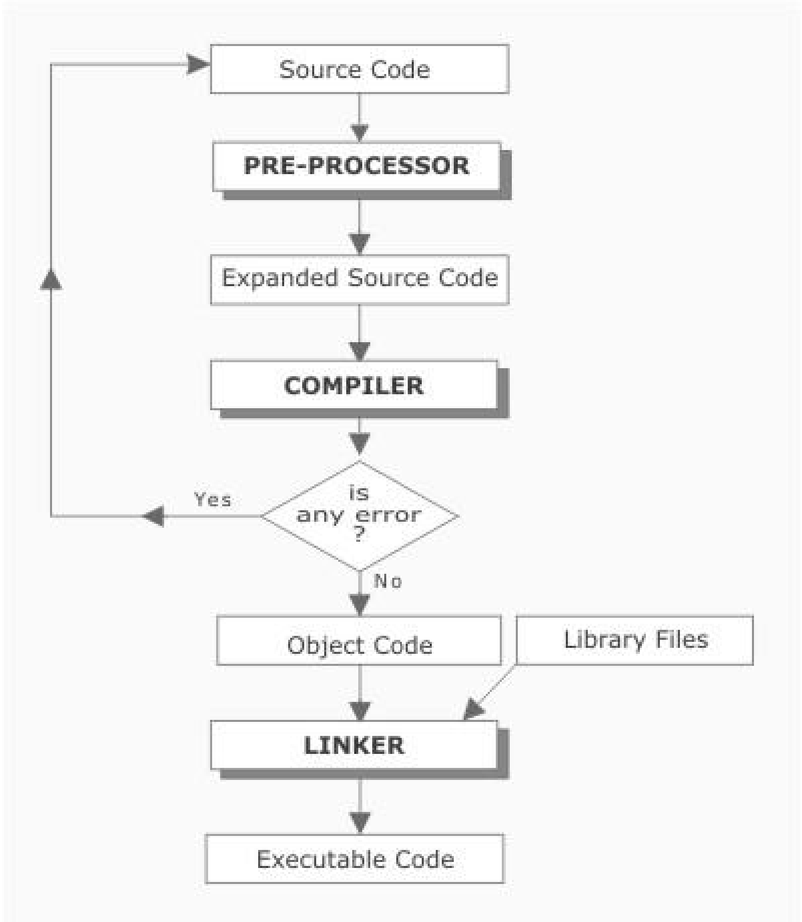
1.5Application of C++

C++ is a versatile language for handling very large programs. It is a suitable for virtually any programming task including development of editors, compilers, database, and communication systems and any complex real-life application systems.Since C++ allows us to create hierarchical-related objects, we can build special object-oriented libraries which can be used later many programmers.While C++ is able to map the real-world problem properly, the C part of C++ gives the languages the abilities to get the machine-level details. C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.It is expected that C++ will replace C as a general-purpose language in the near future.

1.6 Getting Started with C++

A C++ program is a combination of a group of instructions. Instructions are made up of constants, variables, and keyword.

The best way to start learning a programming language is by writing a program



Compilation and Linking of a C++ Program

Integrated Development Environment

Above Figure illustrates the process of translating a C++ source file into an executable file. You can perform entire process of invoking the preprocessor, compiler, and linker with a single action by using Integrated Development environment. These environments consist of a text editor, compiler, debugger, and other utilities integrated into a package with a single set of menus. Preprocessing, compiling, linking, and even executing a program is done by selecting a single item from a menu.

Structure of a Simple C++ PROGRAM

<u>Source code</u>

```
// My first program in C++
 #include <iostream.h>
int main ()
 {
cout<< "Hello World!"; return 0;
 }
```

<u>Output</u>

```
 Hello World!
```

Explanation of the program above

// my first program in C++

All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program **is.**

#include <iostream.h>

Lines beginning with a pound sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor.

In this case the directive #include <iostream.h> tells the preprocessor to include the iostream standard file. This specific file (iostream.h) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program

int main ()

The main function is the point by where all C++ programs start their execution. It does not matter whether there are other functions with other names defined before of after it.

The instructions contained within this function's definition will always be the first ones to be executed in any C++ program.

For that same reason, it is essential that all C++ programs have a main function. The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

{}

The body of the main function is enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout<< "Hello World";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

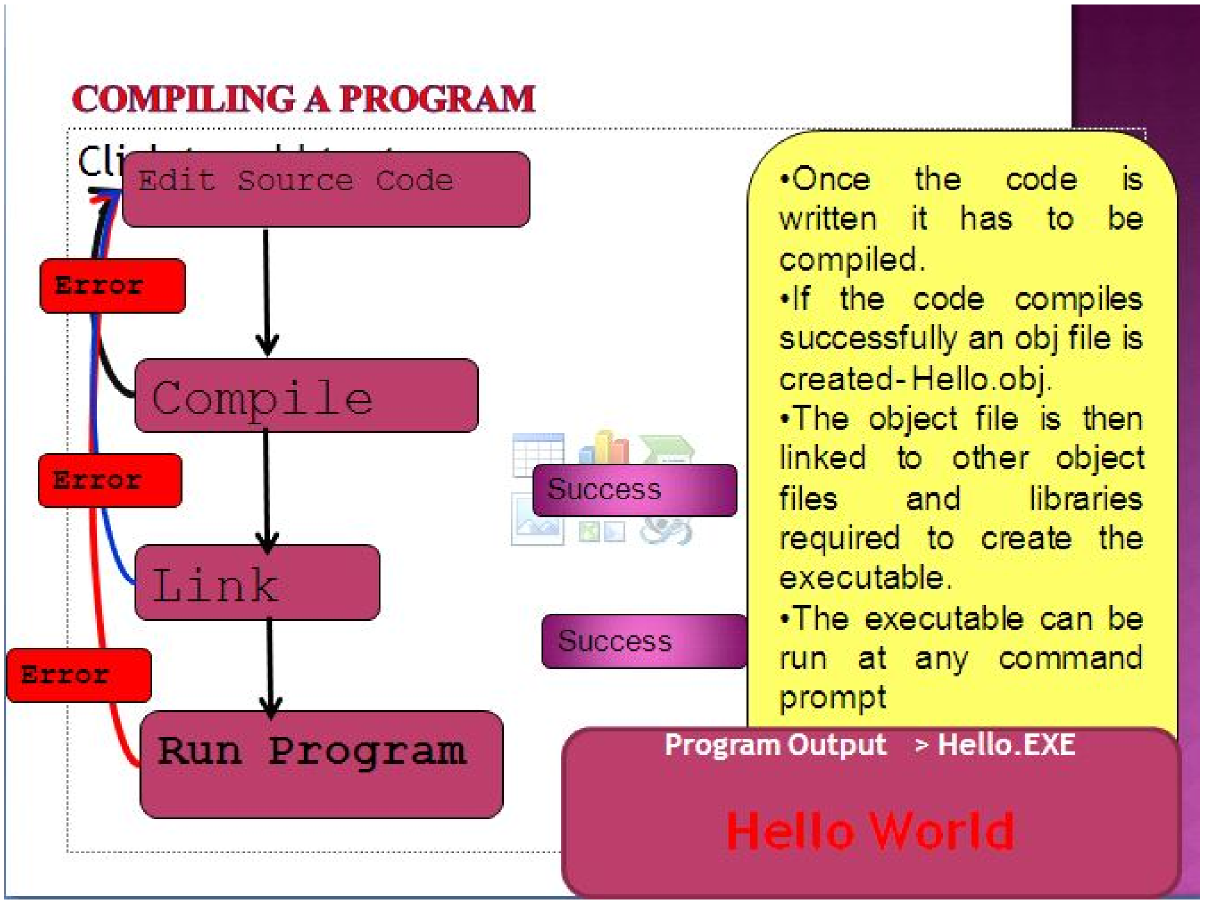
cout is declared in the **iostream** standard file

The statement ends with a semicolon character (;). This character is used to mark the end of the statement.

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0).

A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ program.



COMPILING A PROGRAM

- Once the code is written it has to be compiled.
- If the code compiles successfully an obj file is created- Hello.obj.
- The object file is then linked to other object files and libraries required to create the executable.
- The executable can be run at any command prompt

Program Output > Hello.EXE

```
Hello World
```

CHAP 2: VARIABLES AND CONSTANTS

The constants, variables and keyword are mainly made up of alphabets, numbers and special symbols.

The C++ Character set

Characters include alphabets, digits and special symbols.

Alphabets : A,B………YZa,b……….y,z

Digits : 0,1,2,3,4,5,6,7,8,9

Special symbols:

~,’,!@#$%^&*(){}[“:’;?></.,

2.1 CONSTANTS, VARIABLES AND KEYWORDS

A constant is a quantity that does not change, and can be stored in the memory of a computer

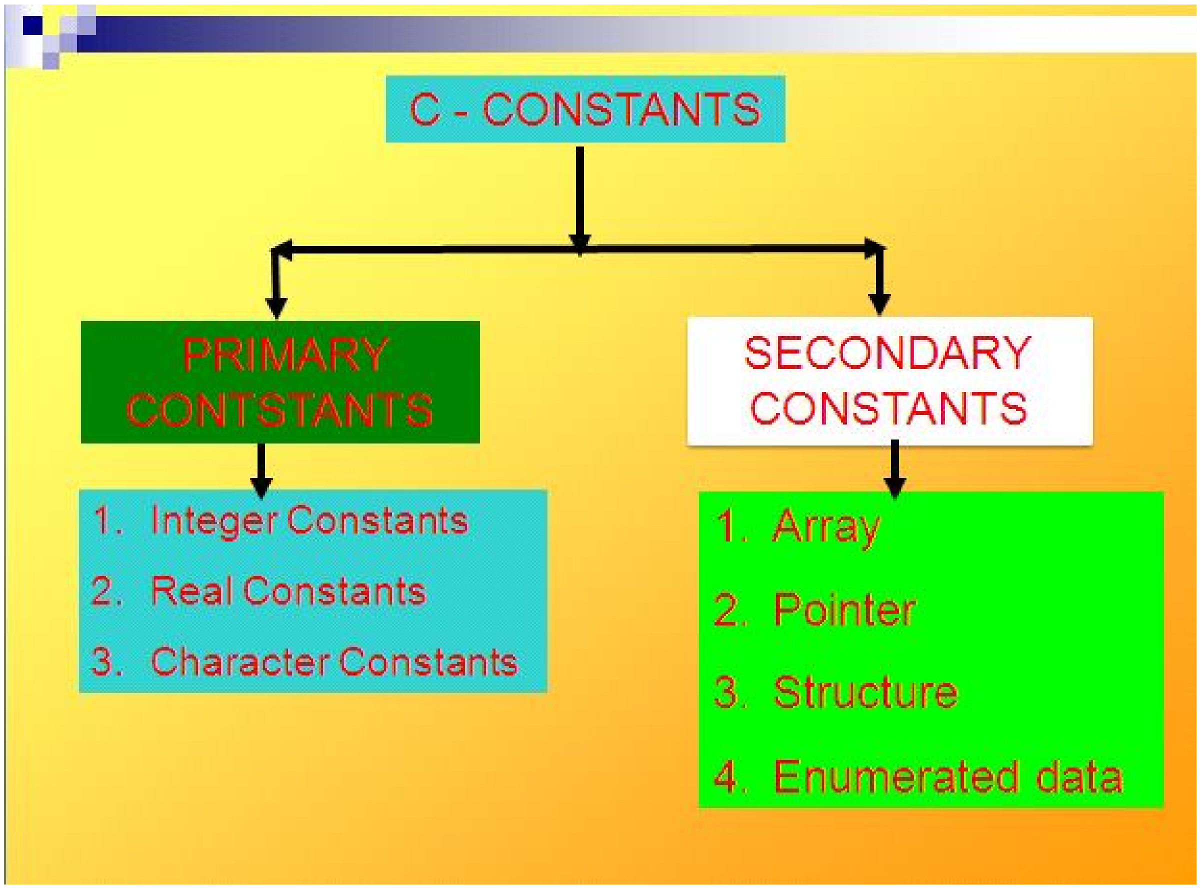
A variable is considered as a name given to the location in memory were the constant is stored

CONSTANTS IN C++

The constants can be divided into two categories:

Primary Constants

Secondary Constants



PRIMARY CONSTANTS

Rules for Integer Constants.

Must have at least one digit

Must not have decimal point

Can be either positive or negative

If no sign precedes the integer its assumed to be positive

No Commas or blanks are allowed within an integer

The range of constant integer is

-32768 to +32767

Rules for real constants

They are also called floating point constants. They can be written in two forms, fractional and exponential form (*4.1e8:- mantissa and exponent parts*)

- Must have at least one digit
- Must have a decimal
- Can be either positive or negative
- Default sign is positive
- No commas or blank are allowed

Rules for character constants

Its either a single alphabet, single digit or single special symbol, enclosed within single inverted commas. Example ‘A’,’5’

The maximum length of character constant can be 1 character.

2.2. VARIABLES IN C++

variable is a memory location where a particular type of data is stored.

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time.

You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5; b = 2; a = a + 1;
result = a - b;
```

The memory in a computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++

VARIABLE TYPES

Char	Character	**1 byte**
int	Integer	**2 bytes**
Float	Floating point number	**4 bytes**

DECLARATION OF VARIABLES

Variables in C++, are declared by first specifying the data type of the variable. followed by a valid variable identifier.

Example:

int a;

float mynumber;

The first one declares a variable of type **int** with the identifier **a**.

The second one declares a variable of type **float** with the identifier **mynumber**. Once declared, the variables **a** and **mynumber** can be used within the rest of their scope in the program

In declaring more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas.

Example:

int a, b, c; This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

int a;

int b;

int c;

PROGRAM EXAMPLE

```
// operating with variables
#include <iostream.h>
int main ()
{
int a, b;
int result;
a = 5; b = 2; a = a + 1;
result = a - b;
cout<< result;
return 0;
}
```

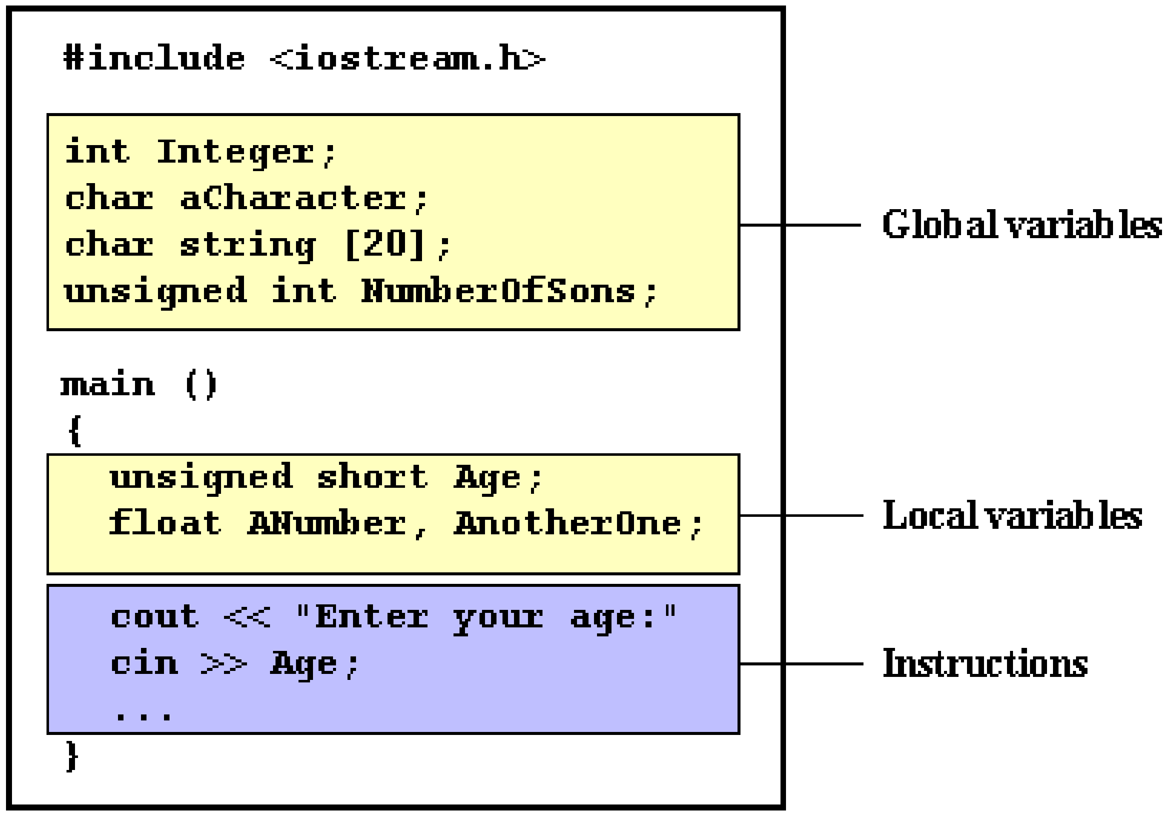
Or

```
#include <iostream>
using namespace std;
int main() { //declare variables of integer type
int x;
int y;
int z; //storing value in variables
x = 25;
y = 10; //adding numbers and store the result in sum z = x + y; //print the result
cout<< "The sum is ";
cout<< z;
return 0; }
```

SCOPE OF VARIABLES

All the variables that are intended to be used in a program must be declared with there type specifier in an earlier point in the code.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



```
#include <iostream.h>

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

main ()
{
   unsigned short Age;
   float ANumber, AnotherOne;

   cout << "Enter your age:"
   cin >> Age;
   ...
}
```

Global variables

Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

Local variables

The scope of **local variables** is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function.
In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

INITIALIZATION OF VARIABLES

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++: The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

Type identifier = initial_value ;

Example: if we want to declare an int variable called a, initialized with a value of 0 at the moment in which it is declared, we could write:

int a = 0;

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

type identifier (initial_value) ;

Example:

int a (0);

Both ways of initializing variables are valid and equivalent in C++.

Program Example

```
// initialization of variables
#include <iostream.h>
int main ()
{
int a=5;
// initial value = 5
int b(2);
// initial value = 2
int result;
 // initial value undetermined
 a = a + 3;
result = a - b;
```

cout<< result; return 0;

}

2.3 DEFINED CONSTANTS (#DEFINE)

You can define your own names for constants by using the #define preprocessor directive. Its format is:

#define identifier value

Example:

#define PI 3.14159265

#define NEWLINE '\n‘

This defines two new constants: PI and NEWLINE.

Once they are defined, you can use them in the rest of the code as if they were any other regular constant,

2.4 INPUT/OUTPUT (I/O)

The standard C++ library includes the header file iostream, which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. The following C++ stream objects can be used for the input/output purpose.

Coutconsole output **cin**console input

cout object

cout is used to print message on screen in conjunction with the insertion operator <<

cout<< "Hello World"; *// prints Hello world on screen*

cout<< 250; *// prints number 250 on screen*

cout<< sum; *// prints the content of variable sum on screen*

To print constant strings of characters we must enclose them between double quotes (").

If we want to print out a combination of variables and constants, the insertion operator (<<) may be used more than once in a single statement

cout<< "Area of rectangle is " << area << " square meter" ;

If we assume the area variable to contain the value 24 the output of the previous statement would be: Area of rectangle is 24 square meter.**cin object** cin can be used to input a value entered by the user from the keyboard. However, the extraction operator >> is also required to get the typed value from cin and store it in the memory location. Let us consider the following program segment: int marks;

```
cin>> marks;
```

In the above segment, the user has defined variable marks of integer type in the first statement and in the second statement he is trying to read a value from the keyboard.

```
// input output example
#include <iostream>
using namespace std;
int main ()
{
int length;
int breadth;
int area;
cout<< "Please enter length of rectangle: ";
cin>> length;
cout<< "Please enter breadth of rectangle: ";
cin>> breadth;
area = length * breadth;
cout<< "Area of rectangle is " << area;
return 0;
}
```

Output :

```
Please enter length of rectangle: 6 Please enter breadth of rectangle: 4 Area of rectangle is 24
```

You can also use cin to request more than one input from the user:

```
cin>> length >> breadth;
```

is equivalent to:

```
cin>> length;
cin>> breadth;
```

cin and strings

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin>>mystring;
```

However, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction.

for example if we want to get a sentence from the user, this extraction operation would not be useful. In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin:

```
// cin and strings
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string name;
cout<< "Enter your name";
getline (cin, name);
cout<< "Hello " << name << "!\n";
return 0;
}
```

Output

```
Enter your name :   Parfait  Hello Parfait!
```

CHAP 3: ARRAYS

3.1 DEFINITION:

An array is a group of logically related data items of the same data-type addressed by a common name and all the items are stored in a contiguous (physically adjacent) memory location.

```
#include<iostream.h>
main()
{
          int x;
          x=5;
          x=10;
          cout<<"The value of x is"<<x;
 }
```

The program above will print the value of x to be 10, because when the value 10 is assigned to x, the earlier value of x is 5 is lost. Ordinary variables are only capable of holding only one value at a time.

There are situations in which you could like to store more than one value at a time in a single variable.

Example:

arranging the percentage marks obtained by 100 students in ascending order.

This can be done in two ways:-

Declaring 100 variables to store percentage marks obtained by 100 different students ie each variable will contain one student’s marks.

Declare one variable (called array or subscripted variable) capable of storing or holding all the 100 values.

Thus an array is a collective name given to a group of similar quantities.

These similar quantities may be percentage marks of 100 students, or salaries of 300 employees, or age of 50 employees

The quantities for an array must be similar. Each member of the group is referred to by the position in the group.

3.2.ONE-DIMENSIONAL ARRAY

Declaring of a one dimensional Array:

An array needs to be declared so that the compiler can know what kind of an array it is and how large an array will b

Syntax:

```
data type name[size];
```

Example:

```
int a[5];
char  ch[10];
float real[10];
```

Array Initialization:

int a [5] = {1,2,3,4,5}; 5 elements are stored in an array ‘a’. The array elements are stored sequentially in separate locations.

Example:

The following group numbers represent percentage marks obtained by six students.

```
Per[6] ={48,50,90,58,47,30}
```

In referring to the numbers of the group the counting of elements begin with zero and not 1. Thus the fourth element is referred to as per [3], the fifth element as per[4].

In the example above per[3] refers to 58, and per[4] refer to 47.

The notation can be taken as per[i], where i can take a value 0,1,2,3,4,……………, depending on the position of the element being referred.

An array as said earlier is a collection of similar elements. The similar elements can be all integers or floats, or characters.

NOTE:-all the elements of any array must be of the same type. You cannot have an array of 10 numbers, of which 5 are ints and 5 are floats.

Program using arrays to find the average marks obtained by 3 students

```
#include<iostream.h>
main()
  {
          float avg, sum = 0;
          int i;
          int marks[3];  //array declaration
          for (i = 0; i<=3; i++)
{
cout<<"Enter the marks";
          cin>> marks[i];     //store data in array
}
          for (i = 0; i<=3; i++)
          {
          sum = sum + marks[i];  //read data from an array
          }
                    avg = sum/4;
                    cout<<"The average is"<<"\t"<<avg;
}
```

Program Explanation:

int marks [3];

The int specifies the type of the variable, and a mark specifies the name of the variable.
The number 3, tells how many elements of the type it will be in the array. This number is referred to as the dimension of the array. The bracket indicates to the compiler that, that is an array.

Accessing Elements of an array

All the array elements are numbered starting with 0. Thus marks[2] is not the second element of the array but the third element.
The program above is using i as the subscript to refer to various elements of the array.

Entering Data into an Array

The following section of code taken from the program A1above places data into the array.

```
for(i = 0; i<=3; i++)
{
        cout<<"Enter the marks";
        cin>> marks[i];     //store data in array
}
```

The for loop causes the process of asking for and receiving a student’s marks from the user to be repeated 3 times.
The first time through the loop, i has a value 0, so the cin>> statement will cause the value typed to be stored in the array element marks [0], the first element of the array. This process will be repeated until i become 3.

Reading Data from an Array

```
for (i = 0; i<=3; i++)
{
        sum = sum + marks[i];  //read data from an array
}
        avg = sum/4;
```

```
cout<<"The average is"<<"\t"<<avg;
```

The code above reads data back from the array and uses it to calculate the average. The for loop causes each students marks to be added to a running total stored in a variable called sum.

When all the marks have been added up the result is divided by 4 the number of students to get the average.

Array Initialization

You can initialize arrays, by storing values in them, during declaration.

Example:

```
int num[6]={2,4,6,8,3,6};
int[n]={5,6,4,3,8,1,4};
float press[]={1.3,4.6,5,7,9.8,5.7};
```

Array elements in memory

Consider the following array declaration:

```
int arr[8];
```

What happens in memory when this declaration is made in memory?. 16 bytes of memory space is reserved. 16 is reserved because each of the 8 integers would be 2 bytes long.

Passing array elements to a function:

Array elements can be passed to a function by calling the function by value or by reference.
In the call by value the values of the array elements are passed to the function, while in the call by reference the address of the array elements are passed to the function

Program to demonstrate call by value

```
#include<iostream.h>
 display(int m);
 main()
{
        int i;
        int marks[]={55,65,75,56,78,78,90};
        for(i =0; i<=6; i++)
        display(marks[i]);
}
 display(int m)
{
                        cout<<m<<endl;
}
```

Program Explanation:

Individual array elements are passed to the function display() at a time. Since at each time its only one element that is passed, its collected in an ordinary variable m, in the function display()

Program to demonstrate call by reference

```
#include<iostream.h>
 disp (int *n);
 main()
{
        int i;
        int marks[] = {55,65,75,56,78,78,90};

        for( i= 0; i<=6; i++)
                disp(&marks[i]);
}
 disp (int *n)
{
        cout<< *n;
    }
```

In this program individual addresses of array elements are passed to the function display();.
The variable in which the address is collected (n) is declared as a pointer variable. And since n contains the address array element, to display out the array elements you need to use value at address operator (*).

3.3. Multidimensional Array

Arrays can have two or more dimensions. The two dimensional array is also called a matrix. In a two dimensional array two subscripts are required to access each element.

Definition

A multi dimensional array is defined as

Syntax:

Data-type array-name [s1][s2]………[sn];

Example:

int axis[3][3][2];

Defines a three- dimensional array with the array-name axis.

The general format for defining a two-dimensional array is:

Syntax:

Data-type array-name [*row-size*] [*column-size*];

Example:

Int marks[4] [3];

Float b[3] [3];

Defines arrays named marks and b respectively. The expression marks [0] [0] will access the first element of the matrix marks and marks [3] [2] will access the last row and last column. The expression accesses the 3rd row and 2nd column element of the b matrix.

Accessing two dimensional array elements

The elements of a two dimensional array can be accessed by the following statement

Marks [i][j];

Where i refers to the row number and j refers to the column number.

Program to illustration addition and subtraction of matrices:

```
        #include<iostream.h>
        main()
        {
                int a[5][5],b[5][5],c[5][5];
                int i,j,m,n,p,q;

                cout<<"Enter row and column size of A matrix";
                cin>>m >>n;
        cout<<"Enter row and column size of B matrix";
                cin>>p >>q;
    if ((m ==p) && (n == q))//check if matrices can be added
                {
                        cout<<"Matrices can be added or subtracted.....\n";
                        //Read matrix A
                cout<<"Enter matrix A elements....\n";
                        for (i = 0; i<m; i++)
                                for(j=0; j<n; ++j)
                                        cin>>a[i][j];

                //Read matrix B
                        cout<<"Enter matrix B elements....\n";
                                for(i =0; i<p; i++)
                                for(j=0;j<q; j++)
```

```
                                        cin>>b[i][j];
    //Addition of two matrices c = a + b
                                  for(i=0; i<m; i++)
                                        for(j=0; j< n; j++)
                                              c[i][j] = a[i][j] + b[i][j];

                            //Print the summation
                            cout<<"Sum of A and B matrices.....\n";
                            for(i =0; i<m; i++)
                            {
                                  for(j=0; j< n; ++j)
                                        cout<< c[i][j]<< " ";
                                  cout<<endl;
                            }
//Subtracting of Matrices: c= A-B
                      for(i=0; i< m; i++)
                            for(j=0; j<n; j++)

                                  c[i][j]=a[i][j] - b[i][j];
                            //Printing matrix subtraction result

                      cout<<"Difference of A and B matrices...\n";
                      for(i=0; i<m; ++i)
                      {
                            for(j=0; j< n; j++)
                            {
                                  cout.width(2);
                                  cout<<c[i][j]<<" ";
                            }
                            cout<<endl;
                      }
```

Initialization at Definition

- A two dimensional array can be initialized during its definition as follows:-

Syntax

```
Data-type matrix-name[row-size][col-size]=
{
                        {elements of first row},
                        {elements of second row},
                        ……..
                        { elements of n-1 row}
};
```

Example:

```
int a[3][3] =
{
        {1,2,3},
        {4,3,1},
        {3,1,2}
};
```

Program to find the minimum value and its position

```
    #include<iostream.h>
#define MAX 5
main()
    {
            int a[MAX],i,min;
            int pos = 0;
            cout<<"Enter the array elements \n";
```

```
        for(i=0;i<MAX;i++)
        {
                cin>> a[i];
                min = a[0];
        }
        for(i=1;i<MAX;i++)
                if (a[i] < min)
                {
                        min= a[i];
                        pos = i;  //fixing the minimum value position
                }
                cout<<"MINIMUM VALUE ="<<min;
                cout<<"POSITION=" <<pos;
}
```

CHAP 4: THE USE OF CONDITIONS

With the introduction of control structures a new concept of the *compound-statement* or *block* is introduced.

A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }.

{statement1;statement2;statement3;}

The control structures include:

If statement

If-else statement

Nested if-else statement

4.1. THE IF STATEMENT

The if keyword is used to execute a statement or block only if a condition is fulfilled.

Syntax:

```
                    if (condition) //no semicolon
                              statement.
```

The if statement is used to make a decision. The block of statements following the if executes if the decision is true, and the block does not execute otherwise.

Example:

If the light is green go

If the light is red stop

Example One:

```
# include<iostream.h>
main()
{
int  age = 21;
if (age = = 85)
  {
cout<<" You have lived through a lot ";
  }
}
```

The program doesn’t do anything if the age is not equal to 85.

What will happened if the expression if (age = 85) is used

Example 2

```
if (sales > 5000)
{
          bonus = 500;
}
```

If sales contains more than 5000 the next statement that is executed is the one in
The block that initializes bonus. If bonus contains less than 5000 the block will
Not be executed.

Example 3

```
if (age <= 21)
```

```
{
cout<<“You are aminor.\n”;
cout<<“What is your grade?”;
cin>> grade;
}
```

If the variable age is less than or equal to 21

Print you are a minor to the screen and go to a new line

Print what is your grade? To the screen

And accept an integer from the keyboard.

If the value of age is greater than 21 the block is skipped.

Assignments (Class Work)

While purchasing certain items a discount of 10% is offered if the quantity purchased is more than 1000. If the quantity and price per item are input through the keyboard, write a program to calculate the total expenses

Solution

```
#include <iostream.h>
main()
 {
        int qty,dis=0;
        float rate,tot;
        cout<< "enter the quantity and rate";
        cin>>qty;
        cin>>rate;
        if (qty > 1000)
        {
                dis = 10;
        tot = (qty * rate) -(qty *rate * dis/100);
        cout<< "Total expenses" << tot;
        }
 }
```

Example of a sample interaction with the program

```
Enter quantity and rate 1200 15.50
Total expenses = frw 16740.000000
Enter quantity and rate 200 15.50
Total expenses = frw 3100.000000
```

In the first run of the program, the condition evaluates to true, since 1200 (value of quantity) is greater than 1000. Therefore, the variable dis which was earlier set to 0, now gets a new value 10 . And using these new value total expenses are calculated and printed.

In the second run the condition evaluates to false, since 200 (the value of quantity) isn't greater than 1000, Thus dis which earlier was set to 0 remains 0, and hence the expression after the minus sign evaluates to zero thereby offering no discount.

4.2THE IF-ELSE STATEMENT

Syntax:

```
        If (condition)
         {
           A block of 1 or more c++ statements
         }
        else
 {
   A block of 1 or more c++ statements
         }
```

If the condition is true, the first part of the if-else is executed, however if the condition is false the block of the c++ statements following the else executes.

Examples:

The following program asks a number from the user. The program then prints a line indicating if a number is greater than zero or that it is not.

```
#include<iostream.h>
main()
 {
        int num;
        cout<<"Enter  your number";
        cin>> num;
        if (num > 0)
        {
                cout<< " More than 0 \n";
        }
        else
        {
                cout<<"Less or equal to 0 \n";
        }
  // No matter what the number was the following statement is      executed
        cout<<"\n\n Thanks for your time\n";
  }
```

Example:

In a company an employee is paid as under:-

If his basic salary is less than frw 1500, then HRA = 10% of basic salary and TA=90% of basic. If his salary is either equal to or above frw 1500, then HRA= frw 500 and TA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

solution

```
# include <iostream.h>
main()
{
        float bs,gs,ta,hra;
        cout<<"Enter the  basic salary";
        cin>> bs;
        if (bs <= 1500)
        {
                hra = bs * 10/100;
                ta = bs * 90/100;
        }
        else
        {
                hra = 500;
                ta = bs * 90/100;
        }
        gs = bs + hra + ta;
        cout <<" The gross Salary is"<<gs;
}
```

Nested If –else statements

The nested if/else structure test for multiple cases by placing if/else selection structure inside the if/else selection structure.

Pseudo code:

```
If student's grade is greater than or equal to 90
        Print "A"
Else
        If student's grade is greater than or equal to 80
                Print "B"
        Else
                If student's grade is greater than or equal to 70
                        Print "C"
                Else
                    If student's grade is greater than or equal to 60
                          Print "D"
                        Else
                                Print "F"
```

C++ Code

```
        If (grade >= 90)
                Cout <<"A";
```

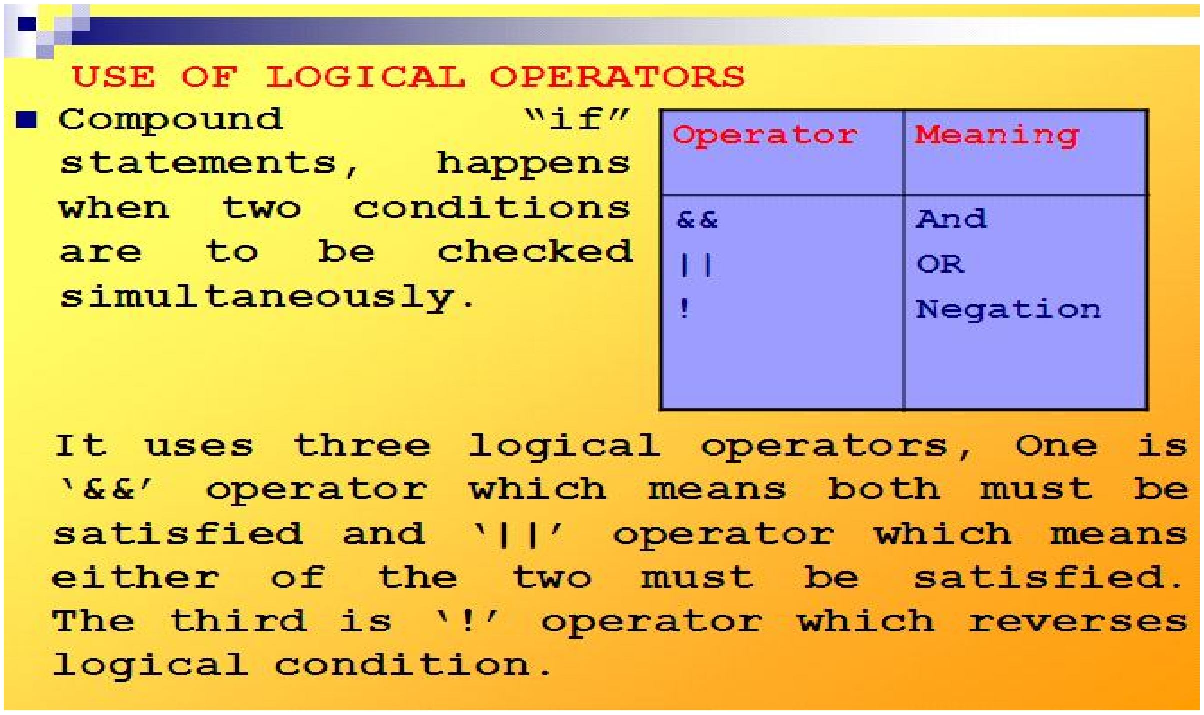
```
          Else if (grade >= 80)
                    Cout <<"B";
          Else if (grade >= 70)
                    Cout <<"C";
          Else if (grade >= 60)
                    Cout <<"D";
          Else
          {
              Cout <<"E";
             Cout <<"You must repeat this course";
            // The braces  help in printing the last  statement.
          }
```

Program example1

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

          cout<<"Enter either 1 or 2 ";
          cin>> i;
          if ( i = = 1)
                    cout <<" You are in Kigali";
          else
          {
                    if(i = = 2)
                              cout <<"You are in Kampala";
                    else
                              cout <<" You must be in TZ";
          }
}
```

Program example2

```
#include<iostream>
using namespace std;
int main()
  {
          float avg, sum = 0;
          int i;
          int marks[3];  //array declaration
          for (i = 0; i<3; i++)
{
cout<<"Enter the marks";
                    cin>> marks[i];     //store data in array
          }
          for (i = 0; i<3; i++)
          {
          sum = sum + marks[i];  //read data from an array
          }
                    avg = sum/3;
                    cout<<"The average is"<<"\t"<<avg<<"\n";
                    if (avg >= 80)
{
          cout<< "Your grade is A";
}
else if (avg >= 70)
{
          cout<< "Your grade is B";
}
else if (avg >= 60)
{
          cout<< "Your grade is C";
}
else if (avg >= 50)
{
        cout<< "Your grade is A";
}
else
{
        cout<<" your grade is E";
}
}
```



USE OF LOGICAL OPERATORS

- Compound "if" statements, happens when two conditions are to be checked simultaneously.

Operator	Meaning
&&	And
||	OR
!	Negation

It uses three logical operators, One is '&&' operator which means both must be satisfied and '||' operator which means either of the two must be satisfied. The third is '!' operator which reverses logical condition.

Example

```
If((a>b) &&(b>c))
  d=a+b;
Else
  d=a-b;
```

Program Example:

The Marks obtained by a student in 5 different subjects are input through the keyboard. The students get a division as per the following rules.

Percentage above or equal to 60– First division

Percentage between 50 and 59 - Second Division

Percentage between 40 and 49 - Third division

Percentage less than 40 - Fail.

Write a program to calculate the division obtained by the student

```
# include<iostream.h>
main()
{
          int m1,m2,m3,m4,m5, per;
          cout<<"Enter the marks in five subjects";
          cin>> m1;
          cin>> m2;
          cin>> m3;
          cin>> m4;
          cin>> m5;
          per = (m1+m2+m3+m4+m5)/5;
          if (per >=60)
                    cout<<"First division";
          else
            {
          if (per >=50)
          cout<<"Second Division";
          else
                  {
                    if (per >=40)
                    cout<<"Third Division";
                    else
                    cout<<"Fail";
                    }
             }
}
```

The problems with the above program are:

Indentation increases as the program increases

Care is needed to match the if's and the else's

Care in the match of braces

All this problems can be eliminated by the usage of 'Logical operators'. The following program illustrates this:-

```
# include<iostream.h>
main()
{
        int m1,m2,m3,m4,m5;
        float per;
        cout<<" Enter the marks for the five subjects";
        cin>>m1 >>m2>> m3 >>m4 >>m5;
        per =(m1+m2+m3+m4+m5)/5;
        if (per >=60)
                cout<< "First Division";
        if ((per>=50)&&(per <60))
                cout<< "Second Division";
        if ((per >=40) && (per<50))
                cout<<"Third Division";
        if (per <40)
                cout<<"Fail";
 }
```

Hierarchy of logical operators

Operators	Type

}

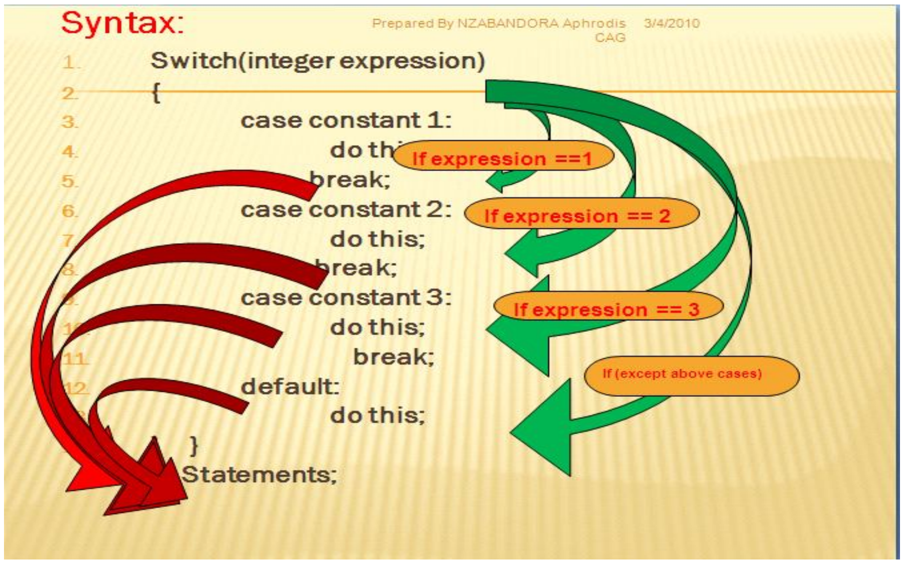
!	Logical NOT
***/%**	Arithmetic and Modulus
+-	Arithmetic
<><= >=	Relational
== !=	Relational
&&	Logical AND
||	Logical OR
=	Assignment

4.3 CASE CONTROL STRUCTURES

The switch statement allows making of decisions from a number of choices

Syntax:

```
                    Switch(integer expression)
                    {
                              case constant 1:
                                        do this;
          break;
                              case constant 2:
                                        do this;
          break;
                              case constant 3:
                                        do this;
          break;
                              default:
                              do this;
                    }
```



Syntax:

Prepared By NZABANDORA Aphrodis 3/4/2010 CAG

```
Switch(integer expression)
{
        case constant 1:
                do this;
                break;
        case constant 2:
                do this;
                break;
        case constant 3:
                do this;
                break;
        default:
                do this;
    }
Statements;
```

The integer expression following the keyword switch is evaluated. The value it gives is then matched, one by one, against the constant value that follows the case statements. When a match is found, the program executes the statement following that case.

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement.

When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

Switch example:

```
switch (x)
 {
case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

if-else equivalent

```
if (x == 1)
 {
cout<< "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
cout << "value of x unknown";
}
```

if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block.

Program Example 1:

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

```
main()
{
        int i=1 ;
        switch (i)
        {
        case 1:
                cout<<"Case One";
                        break;
        case 2:
                        cout<<"Case Two";
                        break;
        case 3:
                        cout<<"Case Three";
                        break;
                default:
                        cout<<"Case Default";
        }
}
```

Program example

```
#include <iostream.h>
main()
{
        int i;
        cout<< " Main Menu"<<endl;
        cout<< "1.  Kenya"<<endl;
        cout<< "2.  Rwanda"<<endl;
        cout<< "3.  Tanzania"<<endl;
        cout<< "4.  Uganda"<<endl;
        cout<< "5.  Exit"<<endl;
        cout<<"Enter your choice"<<endl;
```

```
		cin>> i;
		switch(i)
		{
			case 1:
				cout<< "History of Kenya";
				break;
			case 2:
				cout<< "History of Rwanda";
				break;
			case 3:
				cout<< "History of Tanzania";
				break;
			case 4:
				cout<< "History of uganda";
				break;
			default:
				cout<< "Exit the program";

		}
	}
```

Example:

DEFINED CONSTANTS

```
// defined constants: calculate circumference

#include <iostream.h>



#define PI 3.14159
#define NEWLINE '\n';

int main()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

  return 0;
}
```

CHAP 5: LOOP CONTROL STRUCTURES

While Loop

Do-While Loop

For Loop

Nested for Loop

Loops involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied.

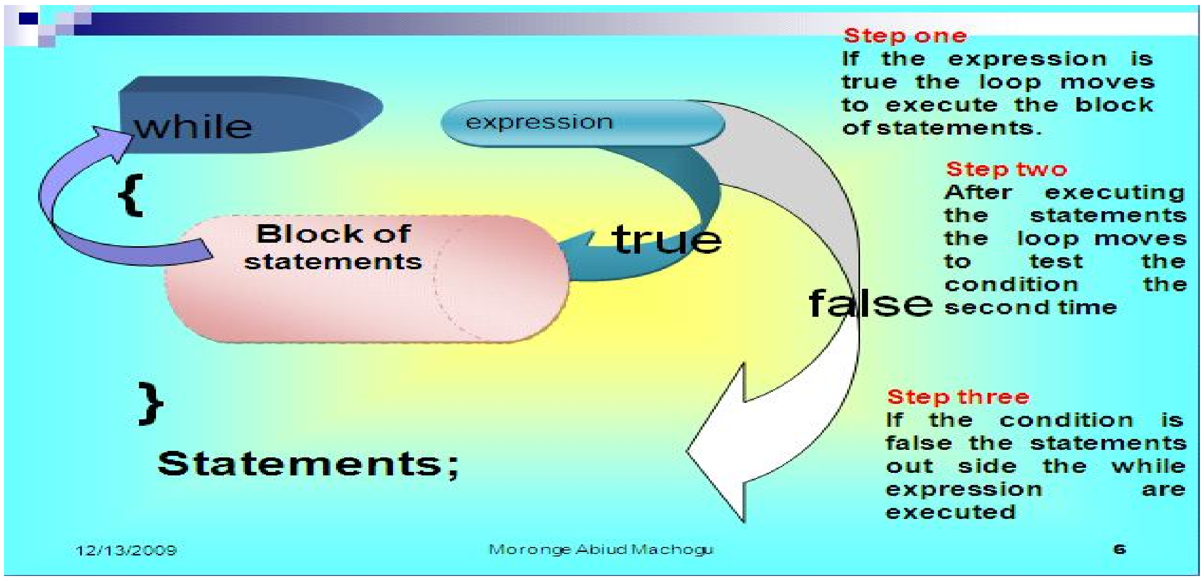
While statement

The While loop assists in the program where you want to carry out an activity for a certain number of times.

Example:- calculating gross salaries of ten people or converting temperatures from centigrade to Fahrenheit for 15 different cities

Syntax:

```
While (condition)
{
        Statement;
}
```



Step one
If the expression is true the loop moves to execute the block of statements.

Step two
After executing the statements the loop moves to test the condition the second time

Step three
If the condition is false the statements out side the while expression are executed

PROGRAM EXAMPLE:

```
#include <iostream.h>
main()
{
        int p,t,count;
        float r,si;
        count = 1;
        while (count <=3)
        {
        cout<<"The the principle, rate and time";
                cin>> p;
                cin>> r;
                cin>> t;
                si = p * r * t;
        cout<< "Simple Interest " << si <<endl;
        count = count++ ;
        }
}
```

The condition to be tested may use the relational operators:

Example:-

```
        While (I <= 10)
        While (I <=10 && j <=15)
        While (j >10 && (b <15 || c<20)
```

Increment statements

Example A:

```
main()
{
        int i;
        while (i++ <10)
                cout<<I;
}
```

In the statement while (i++ < 10), first the comparison of value of I with 10 is performed and then the increment of I takes place. When the control reaches cout, I has already been incremented, hence i must be initialized to 0;

Example B:

```
        main()
        {
                int i=0;
                while (++I <=10)
                        cout<<I;
```

In the statement while(++I <= 10), first incrementation of I takes place, then the Comparison of value of I with 10 is performed.

Assignment 1 (Class work)

Program to calculate the average of 5 marks using the while loop. The marks are entered from the keyboard

```
#include<iostream.h>
main()
{
        int i, sum = 0, count = 0, marks;
```

```
          cout<<"Enter the marks -1 at the end.....\n";
          cin>> marks;
          while ( marks != -1)
          {
                    sum += marks;
                    count++;
                    cin>>marks;
          }
          float avg = sum / count;
          cout<<"The average is "<<avg;
}
```

<u>The do-While Loop</u>

Syntax:

```
do
          {
                    this;
                    and this;
                    and this;
                    and this;
          }while (this condition is true);
```

The do-While loop execute its statements at least once even if the condition fails for the first time. The usefulness of the do….while construct

There are cases in which the user has to be prompted to pres In s m (male) or f (female). If the user make a mistake and presses a different key the message needs to be shown again and then the user should be allowed to re-enter one of the two options.

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

```
{
        char inchar;
        do
        {
                cout<<"Enter your sex (m/f):";
                cin>> inchar;
        }while (inchar != 'm' && inchar != 'f');
        if (inchar == 'm')
                cout<<" you are male";
        else
                cout<<"you are female";
}
```

Program Example 1:

```
#include<iostream.h>
main()
{
        while (4<1)
                cout <<”Here there”;
}
```

If the condition fails the cout will not be executed even once.

Program Example 2:

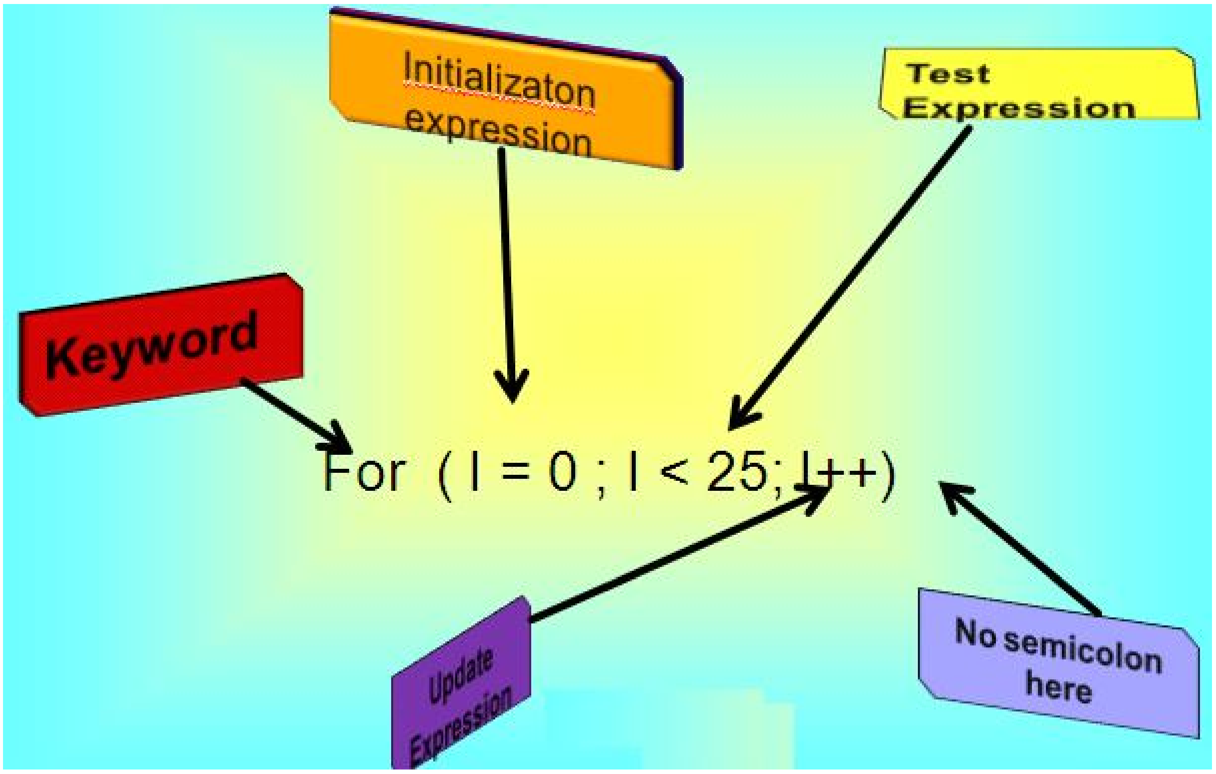
```
#inluce<iostream.h>
main()
{
        do
        {
                cout<<”Hello there”;
        }while(4<1);
}
```

<u>For Loop</u>

The for loop specifies three elements about the loop in one single line.Setting a loop counter to an initial value. Testing the loop counter to determine whether its value has reached the number of repetitions desired .Increasing the value of loop counter each time the program segment within the loop has been executed.

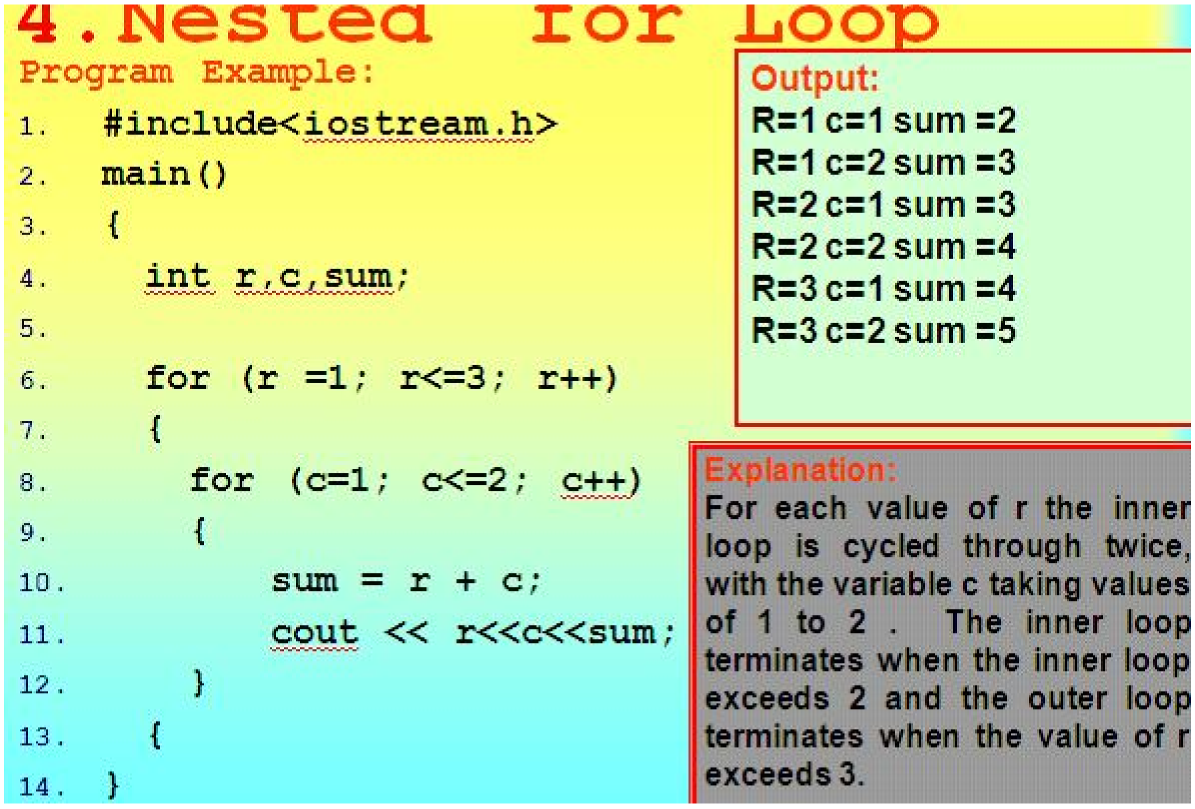
Syntax: for (initialize counter; test counter; increment counter)

```
        {
                do this;
                and this;
                and this;
        }
```



Program example: Calculate the simple interest:

```
#include <iostream.h>
main()
{
        int p,t,count;
        float r,si;
        for (count = 1; count <=3; count = count + 1)
        {
        cout<<"The the principle, rate and time";
                cin>> p;
                cin>> r;
                cin>> t;
                si = p * r * t;
        cout<< "Simple Interest " << si <<endl;
        }
}
```



4.Nested for Loop

Program Example:

```
#include<iostream.h>
main()
{
  int r,c,sum;

  for (r =1; r<=3; r++)
  {
    for (c=1; c<=2; c++)
    {
        sum = r + c;
        cout << r<<c<<sum;
    }
  {
}
```

Output:

```
R=1 c=1 sum =2
R=1 c=2 sum =3
R=2 c=1 sum =3
R=2 c=2 sum =4
R=3 c=1 sum =4
R=3 c=2 sum =5
```

Explanation:
For each value of r the inner loop is cycled through twice, with the variable c taking values of 1 to 2 . The inner loop terminates when the inner loop exceeds 2 and the outer loop terminates when the value of r exceeds 3.

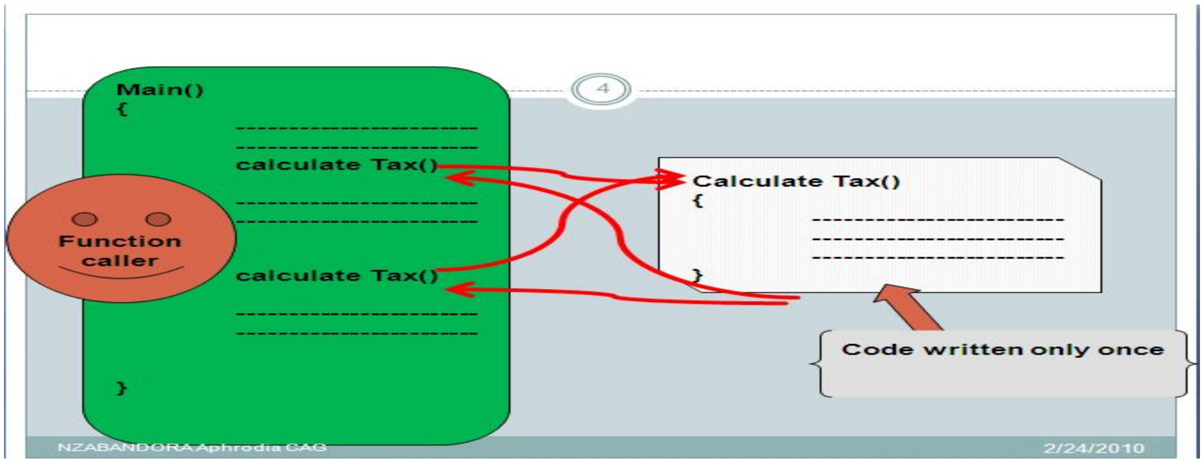
An Odd Loop

```
//Execute a loop an unknown number of times
#include <iostream.h>
main()
{
        char another =’y’;
        int sum;
        while(another == ‘y’)
        {
          cout <<”Enter a number”;
        cin>>num;
        cout<<“Square of “<<num <<num * num;
          cout <<”Do you want to enter another number y/n”;
                cin>> another;
        }
{
```

CHAP 6: THE USE OF FUNCTIONS

6.1 DEFINITION

A function is a block of instructions that is executed when it is called from some other point of the program .Every C++ program has at least one function, main(). When your program starts, main() is called automatically. main() might call other functions, some of which might call still others.



Advantages of functions

Modular Programming

Reduction in the amount of work and development time

Program and function debugging is easier

Division of work is simplified.

Reduction in size of the program

Functions can be accessed repeatedly.

Syntax:

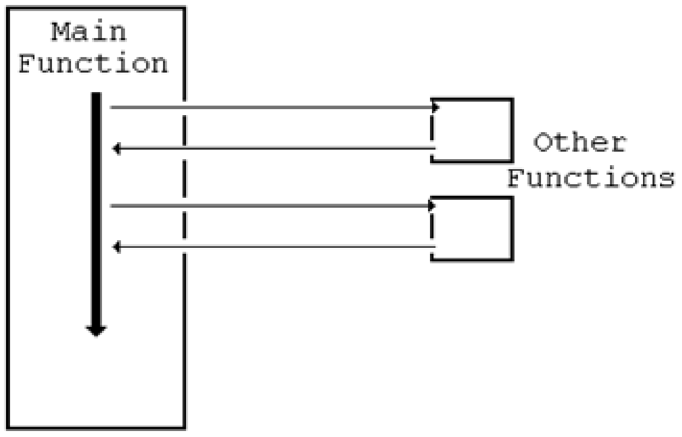
type nameoffunction* (*argument1*, *argument2*, ...) *statement

where:

type is the type of data returned by the function.

name is the name by which it will be possible to call the function.

arguments (as many as wanted can be specified). Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, int x) and which acts within the function like any other variable.
They allow passing parameters to the function when it is called. The different parameters are separated by commas.
·***statement*** is the function's body. It can be a single instruction or a block of instructions. In the latter case it must be delimited by curly brackets **{}**.
When a C++ function is called, the statements in the called function are executed. When the function is finished, control returns to the place from which it was called, and the program continues



Declaring and Defining Functions

In order to use functions in the program you declare and then define the function.
The declaration tells the compiler the name, return type, and parameters of the function. The definition tells the compiler how the function works. No function can be called from any other function that hasn't first been declared. The declaration of a function is called its prototype.

```
void function_name( void )
 {
        // details of what the function does
 }
```

The void inside the parentheses indicates that no information is delivered to the function when it is called and the void before the name indicates that the function returns nothing to the main function.

```
#include <iostream.h>
  // function prototypes
            void say hello ( void )
                        {  cout<<“HELLO”;}
            void say_goodbye( void )
         {cout<<“GOODBYE”;}
void main( void )
  {
            cout<< "This program calls two functions" << endl;
            cout<< "First a greeting... here goes..." << endl;
             say_hello();                  //  calling a function
cout<< "Now back in main" << endl;
cout<< "Now we call the other function..." << endl;
 say_goodbye(); //  calling another function
cout<< "Back in main. That's all folks." <<endl;
}
// definitions of the functions
void say_hello( void )                         // no semicolon here
  {
            cout<< "This is the hello function" << endl;
            cout<< "HELLO THERE!" <<endl;
  }
void say_goodbye( void )                    // no semicolon here
  {
            cout<< "This is the goodbye function" << endl;
            cout<< "GOODBYE, Hope you enjoyed this program" << endl;
 }
```

Program Output.

This program calls two functions First a greeting... here goes... This is the hello function HELLO THERE!" Now back in main Now we call the other function...
This is the goodbye function GOOD BYE, Hope you enjoyed this program Back in main. That's all folks.

Note

The prototypes are read by the compiler and used to introduce (or announce) each function's name, and details (its *signature*).

The function definitions contain the actual instructions (statements) which the function executes. Each definition starts with the function name followed by statements enclosed in a pair of braces {}.

Sample program:

Calculates wages using a function

```
#include <iostream.h>
                                                    // function prototype
  void calc_wages( int hours, float rate );

 void main( void )
 {
  int hours_worked;
  float hourly_rate;

  cout << "Enter the number of hours worked ";
  cin >> hours_worked;

  cout << "Enter the hourly pay rate ";
  cin >> hourly_rate;

  calc_wages( hours_worked, hourly_rate );  //Note 1

 }                                                  // end main

                                                    // function definition
 void calc_wages( int hours, float rate )
 {
  float wage;
  wage = hours * rate;
  cout << "The wage is " << wage << endl;
 }
```

In main, the function calc_wages is *called*. The parameters hours_worked and hourly_rate are *passed to the function*.

That means that the values in these variables are sent to the function.

The parameters hours_worked and hourly_rate are called "actual parameters" because these are the values that are actually used when the function is called.

In the function definition, the parameter names are hours and rate. These parameters are called "formal parameters" because they show the form the parameters must takeThe formal parameters and the actual parameters *don't* have to have the same names, but they must have matching types. The function definitions is put below main() ,and the prototypes above main(). If a function expects any parameters then they must be provided when the function is called.

NOTE:

C++ program is a collection of one or more functions

A function gets called when a function name is followed by a semicolon.

A function is defined when function name is followed by a pair of braces in which one or more statements may be present

Functions can be nested.

A function can be called any number of times.

A function can be called from another function but can not be defined in another function.

A function can call itself.

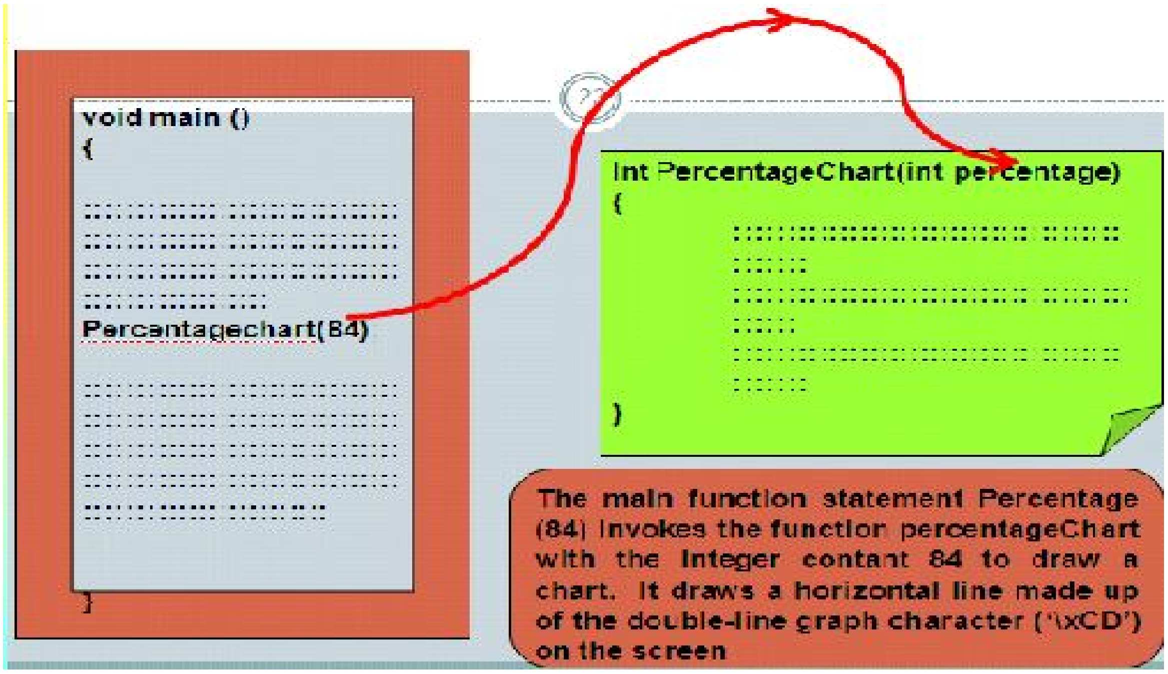
There are two types of functions

-Library functions

-User defined functions.

Passing data within functions

The entity used to convey the message to a function is the function argument, it can be a numeric constant, a variable, multiple variables, user defined data type etc.



void main ()
{
Percentagechart(84)
}

Int PercentageChart(int percentage)
{
}

The main function statement Percentage (84) invokes the function percentageChart with the integer contant 84 to draw a chart. It draws a horizontal line made up of the double-line graph character ('\xCD') on the screen

Function Return Data type

The returned value from a constant can be a variable, a user-defined data structure, a general expression, a pointer to a function or a function call.

Program to find the factorial of a number

```
#include<iostream.h>
int fact( int n)
        {
                float result;
                if(n==0)
                        result = 0;
                else
                {
                        result = 1;
                        for(int i =2; i <= n; i++)
                                result = result * i;
                }
                return result;
}
main()
{
        int n;
        cout<<"Enter the number whose factorial is to be found :";
        cin>>n;
        cout<<"The factorial of " <<n << " is " << fact(n) <<endl;
}
```

Program Explanation

The definition before main() indicates that the function fact takes an integer argument and return an integer data type and placing it in the return statement

When a function has nothing specific to return or take, it is indicated by void, such functions are called void functions

Limitation of return

A key limitation of the return statement is that it cannot return more than one item from a function.

Parameter Passin

Parameter passing is a mechanism for communication of data and information between the calling function and the called function .

C++ supports three types of parameter passing schemes:-

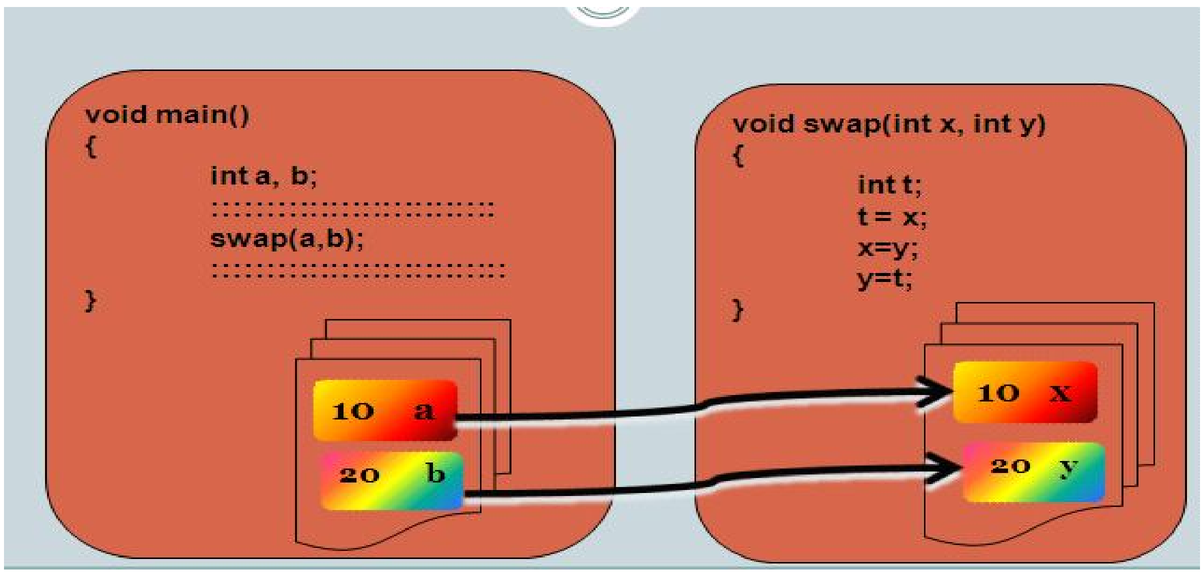
- Pass by Value
- Pass by Address
- Pass by reference

Pass by Value

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

```
void swap(int x, int y)
{
        int t;   //temporary used in swapping
        cout<<" Value of x and y in swap before exchange: "<<x <<"" <<y<<endl;
        t = x;
        x = y;
        y = t;
cout<<" Value of x and y in swap after exchange:"<<x <<"" <<y<<endl;
}
main()
{
        int a , b;
        cout<<"Enter two integers <a, b>:";
        cin>>a >> b;
        swap (a,b);
        cout<<"Value of a and b on swap(a,b) in main():" <<a<<""<<b;
}
```

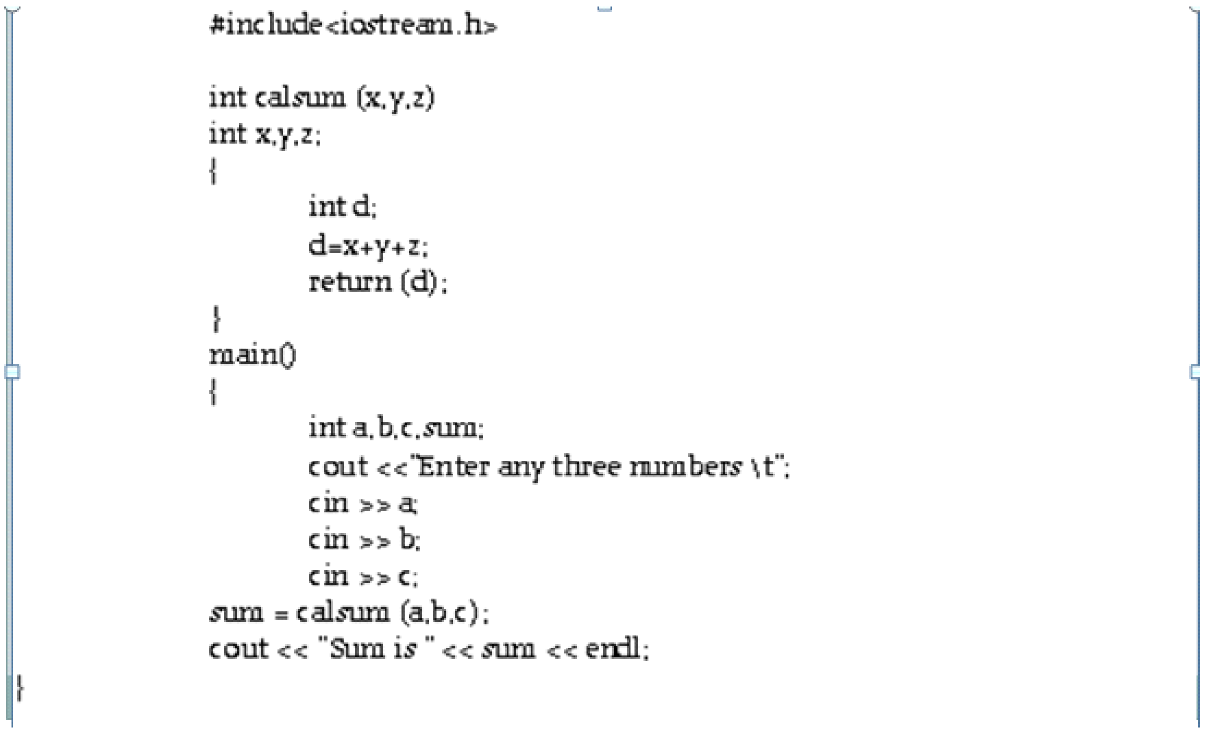
In the main() function, the statement swap(x,y) invokes the function swap() and assigns the contents of the actual parameters a and b to the formal parameters x and y respectively. In the swap() function, the input parameters are exchanged.



```
void main()
{
            int a, b;
            ::::::::::::::::::::::::::
            swap(a,b);
            ::::::::::::::::::::::::::
}
```

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

Program Example



```
#include<iostream.h>

int calsum (x,y,z)
int x,y,z;
{
        int d;
        d=x+y+z;
        return (d);
}
main()
{
        int a,b,c,sum;
        cout <<"Enter any three numbers \t";
        cin >> a;
        cin >> b;
        cin >> c;
sum = calsum (a,b,c);
cout << "Sum is " << sum << endl;
}
```

Explanation:

In main we enter the values **a,b,c,** through the keyboard and then output the sum of **a,b,c**.

The values **a,b,c** are passed on to the function **calsum()** by making a call to the function **calsum()** and mentioning **a,b,c** in the parenthesis.

Sum = calsum(a,b,c)

In the **calsum()** function these values get collected in three variables x,y,z.

```
        Calsum (x,y,z)
        Int x,y,z;
```

The variables a,b,c are called actual arguments and x,y,z are formal arguments.

There are two methods of declaring formal arguments

```
calsum(x,y,z)
int x,y,z;

calsum (int x, int y, int z)
```

Program Example

```
#include <iostream.h>

int addition (int a, int b)

     {
     int r;
       r=a+b;
     return (r);
     }
     int main ()
     {
     int z;
      z = addition (5,3);
     cout<< "The result is " << z;
     return 0;
     }
```

Example:

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

          void showdashes();            // declaration

          main()
          {
                    cout <<" This is to be underlined \n";
                    showdashes();         //function call
                    cout<< " Do that once again \n";
                    showdashes();         //function call
```

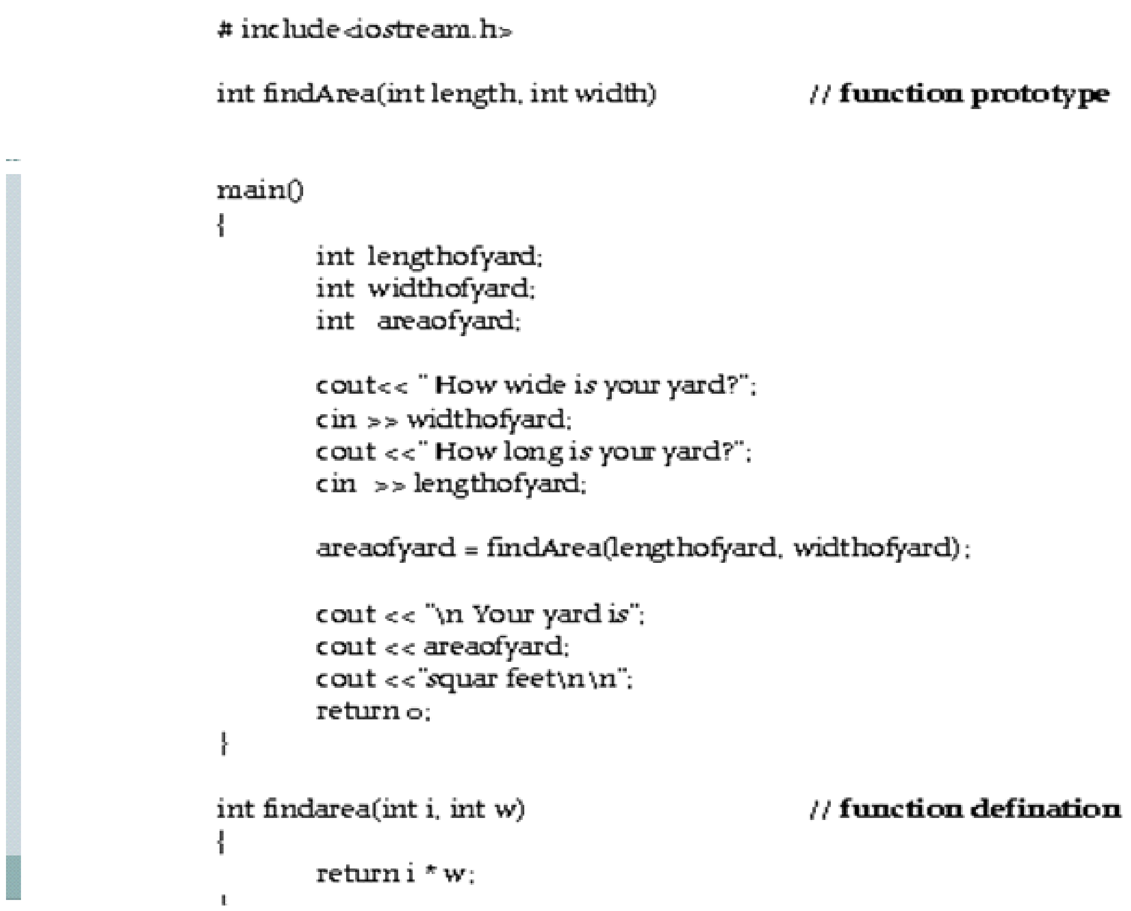
```
}

void showdashes()               //function body
{
cout<<"--------------------------------\n";
}
```

Many of the built in functions have their function prototype already written in the files that are included in the program using the #include. For functions that are created, a prototype must be included.

The function prototype is a statement that ends with a semicolon, it consists of a function's return type, name and parameter list.

Example

Int findArea (int length, int width);The function prototype and the function definition must agree on the return type else an error will be created.



```
# include<iostream.h>

int findArea(int length, int width)                // function prototype


main()
{
        int  lengthofyard;
        int  widthofyard;
        int   areaofyard;

        cout<< " How wide is your yard?";
        cin >> widthofyard;
        cout <<" How long is your yard?";
        cin  >> lengthofyard;

        areaofyard = findArea(lengthofyard, widthofyard);

        cout << "\n Your yard is";
        cout << areaofyard;
        cout <<"squar feet\n\n";
        return o;
}

int findarea(int i, int w)                         // function defination
{
        return i * w;
```

```
    }
```

The name, the return type and the parameter type of the function prototype and the function definition are the same.

The only difference is that the function prototype ends with a semicolon and has no body.

Function Assignments

Variables have a scope, which determines how long it is available to your program.

Variables declared within a block are scoped to that block, they can be accessed only within that block and go out of existence

The variables are of two types

- Local Variable
- Global Variable.

Local Variables:

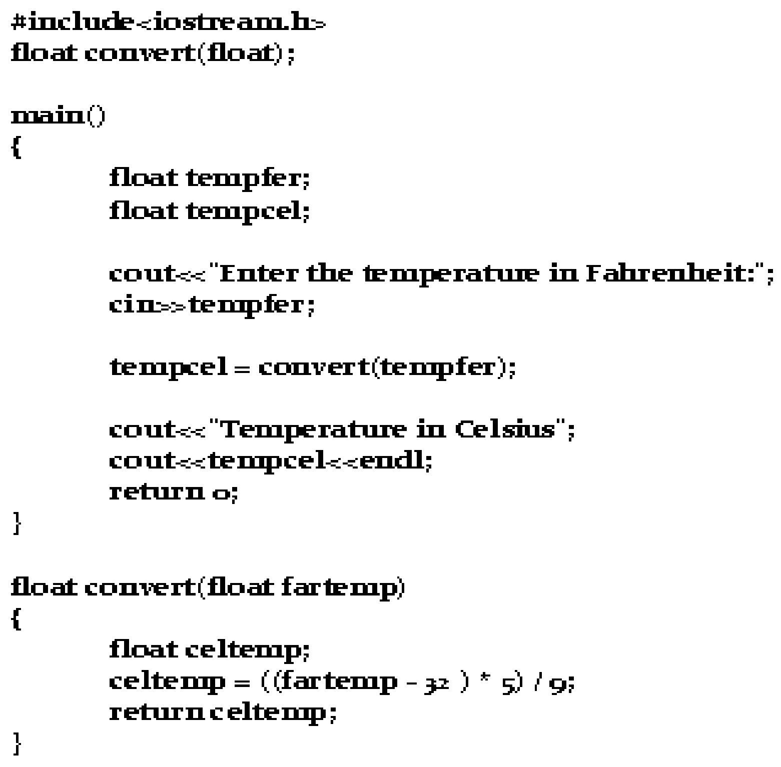
This are variables declared within the body of a function. They exist only locally within the function itself.
The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function.

Program Example:

```
#include<iostream.h>
float convert(float);

main()
{
        float tempfer;
        float tempcel;

        cout<<"Enter the temperature in Fahrenheit:";
        cin>>tempfer;

        tempcel = convert(tempfer);

        cout<<"Temperature in Celsius";
        cout<<tempcel<<endl;
        return 0;
}

float convert(float tempfer)
{
        float tempcel;
        tempcel = ((tempfer - 32 ) * 5) / 9;
        return tempcel;
}
```

The variable tempcel in the function convert is local.

The variables in the function can be renamed as Fertemp and celTemp and the program will work ok.



```
#include<iostream.h>
float convert(float);

main()
{
        float tempfer;
        float tempcel;

        cout<<"Enter the temperature in Fahrenheit:";
        cin>>tempfer;

        tempcel = convert(tempfer);

        cout<<"Temperature in Celsius";
        cout<<tempcel<<endl;
        return 0;
}

float convert(float fartemp)
{
        float celtemp;
        celtemp = ((fartemp - 32 ) * 5) / 9;
        return celtemp;
}
```

Global Variables:

Global variables have a global scope and are available anywhere within the program.

```
#include<iostream.h>

void myfunction();

int x= 5, y=7;

main()
{


        cout<<"x form main"<<x<<"\n";;
        cout<<"y from main"<<y<<"\n";

        myfunction();

        cout<<"x from main"<<x<<"\n";
        cout<<"y from main "<<y<<"\n";

        return 0;
}

void myfunction()
{
        int y = 10;

        cout<<"x from myFunction:"<<x<<"\n";
        cout<<"y from myfunction:"<<y<<"\n";
}
```

The arguments passed in to the function are local to the function.

Changes made to the arguments do not affect the values in the calling function.

```
#include<iostream.h>

void swap(int x, int y);

main()
{
        int x=5, y = 10;

        cout<<"Main. Before swap, x:"<< x <<" y :"<< y <<"\n";

        swap(x,y);

        cout<<"Main. After swap, x: "<< x <<" y :"<< y << "\n";

}

void swap(int x, int y)
{
                int temp;

                cout<<"Swap. Before swap, x:"<< x <<" y :"<< y <<"\n";

                temp = x;
                x = y;
                y = temp;

                cout<<"Swap. After swap, x: "<< x <<" y :"<< y << "\n";
}
```

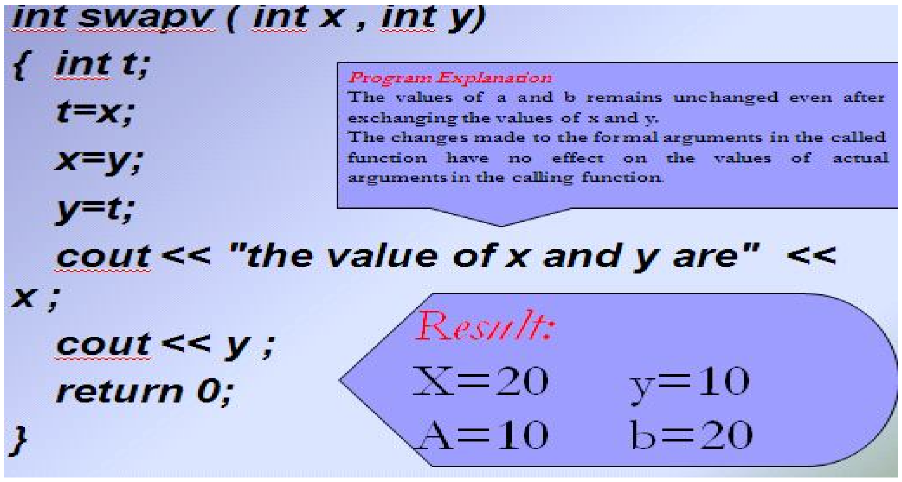
Function Calls

Arguments can be passes to functions in two ways

- Passing values of the argument
- Passing addresses of the arguments (call by reference)

Program Example: Passing values by argument

```
    #include<iostream.h>
    int swapv(int x, int y);
    main()
        {
                int a=10, b = 20;
                swapv(a,b);
                cout<< "\nThe values of a and b are" << a;
        cout<< b <<endl;
        return 0;
}
```



```
int swapv ( int x , int y)
{  int t;
   t=x;
   x=y;
   y=t;
   cout << "the value of x and y are"  <<
x ;
   cout << y ;
   return 0;
}
```

Program Explanation
The values of a and b remains unchanged even after exchanging the values of x and y.
The changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

Result:

```
X=20     y=10
A=10     b=20
```

Program Example: Program to swap any two integers passed to it. passing by reference and declaring function prototype

```
#include<iostream.h>
int swapthem (int num1, num2);
main()
{
        int i =10, j= 20;
        cout<<"Before Swap, i is " << i
                <<" and j is "<< j <<"\n\n";
        swapthem(&i , &j);
        cout<<"\n\n After swap, i is " <<i
                        <<"and j is "<<j<<"\n\n";
        return 0;
}
```

```
int swapthem(*num1,*num2)
{
int temp;   //Variable to hold in between swapped values
    temp =*num1;
    *num1 = *num2;
    *num2 = temp;
    return 0;
```

6.2. OVERLOADING FUNCTIONS:

Function overloading is the creation of more than one function with the same name.
Example:

```
        Int myfunction (int, int);
```

```
Int myfunction (int, float);

Int myfunction(int);
```

Example:

An overloaded add() function handles different types of data as shown below

```
               //Declarations
    Int add(int a, int b);                  //prototype 1
    Int add(int a, int b, int c);           //prototype 2
    Double add(double x, double y);         //prototype 3
    Double add(int q, double p,);           //prototype 4
               //Function calls
               Cout <<add(5, 10);                  //uses prototype 1
               Cout <<add(15, 10.0)     //uses prototype 4
               Cout <<add(12.5, 7.5)    //uses prototype 3
               Cout <<add(5, 10, 15)    //uses prototype 2
```

The unique difference between the overloaded function is the signature/ the parameters in either the number or data type of their arguments.

Program Example:

```
#include<iostream.h>
          void show(int val)
          {
               cout<<" Integer:" <<val<<endl;
          }
          void show(float val)
          {
               cout<< "double: " << val << endl;
          }
          main()
          {
```

```
        show (420);
        show (3.14);
    }
```

Recursion:

Recursion is the process by which a function calls itself.

When a function calls itself a new copy of the same function is run.

The local variables in the second call are independent of the local variables of the first call and they don't affect one another.

Problem 1:

Problem to solve using recursion: Fibonacci series:

1,1,2,3,5,8,13,21,34……….

Each number, after the second, is the sum of the two numbers before it.

Problem 2:

```
//Program to compute the factorial of a number
// The factorial of 5 is 5 * 4 * 3 * 2 * 1 = 120;

#include<iostream.h>

factorial (int x);

main()
{
        int a, fact;

        cout<<"Enter any number";
        cin >> a;

        fact = factorial (a);

        cout <<"Factorial of" <<a<<"is"<<fact<<endl;

}

factorial (int x)
{
        int f = 1, i;

        for (i = x; i >=1; i--)

                        f = f * i;

        return (f);
}
```

CHAP 7: FRIENDSHIP AND INHERITANCE

7.1. FRIEND FUNCTIONS

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.

Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
// friend functions
#include <iostream>
using namespace std;
class CRectangle
{
int width, height;
public:
void set_values (int, int);
int area ()
{
return (width * height);}
friend CRectangle duplicate (CRectangle);
};
void CRectangle::set_values (int a, int b) {
width = a;
height = b;
}
CRectangle duplicate (CRectangle rectparam)
{
CRectangle rectres;
rectres.width = rectparam.width*2;
rectres.height = rectparam.height*2;
return (rectres);
}
int main () {
CRectangle rect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
cout<< rectb.area();
return 0;
}
```

The duplicate function is a friend of CRectangle. From within that function we have been able to Access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
#include <iostream>

using namespace std;

class Box {
   double width;

   public:
      friend void printWidth( Box box );
      void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid ) {
   width = wid;
}

// Note: printWidth() is not a member function of any class.
void printWidth( Box box ) {
   /* Because printWidth() is a friend of Box, it can
   directly access any member of this class */
   cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main() {
   Box box;

   // set box width without member function
   box.setWidth(10.0);

   // Use friend function to print the wdith.
   printWidth( box );

   return 0;
}
```

```
// friend class
#include <iostream>
using namespace std;
class CSquare;
class CRectangle {
int width, height;
public:
int area ()
{return (width * height);}
void convert (CSquare a);
};
class CSquare {
private:
int side;
public:
void set_side (int a)
{side=a;}
friend class CRectangle;
};
void CRectangle::convert (CSquare a) {
width = a.side;
height = a.side;
}
int main () {
CSquare sqr;
CRectangle rect;
sqr.set_side(4);
rect.convert(sqr);
cout<< rect.area();
return 0;
}
```

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member
functions could have access to the protected and private members of CSquare, more concretely to

CSquare::side, which describes the side width of the square.
134
You may also see something new at the beginning of the program: an empty declaration of class
CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as
a
parameter in convert()). The definition of CSquare is included later, so if we did not include a
previous empty declaration for CSquare this class would not be visible from within the definition
of
CRectangle.
Consider that friendships are not corresponded if we do not explicitly specify so. In our example,
CRectangle is considered as a friend class by CSquare, but CRectangle does not consider
CSquare
to be a friend, so CRectangle can access the protected and private members of CSquare but not
the
reverse way. of course, we could have declared also CSquare as friend of CRectangle if we
wanted to another property of friendships is that they are *not transitive*: The friend of a friend is
not consideredto be a friend unless explicitly specified.
Inheritance between classes
A key feature of C++ classes is inheritance. Inheritance allows to create classes which are
derived from other classes, so that they automatically include some of its "parent's" members,
plus its own.
For example, we are going to suppose that we want to declare a series of classes that describe
polygons like our CRectangle, or like CTriangle. They have certain common properties, such as
both can be described by means of only two sides: height and base. This could be represented in
the world of classes with a class CPolygon from which we would derive
the two other ones: CRectangle and CTriangle.
The class CPolygon would contain members that are common for both types of polygon. In our
case: width and height. And CRectangle and CTriangle would be its derived classes, with
specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class.

```
// derived classes
#include <iostream>
using namespace std;
class CPolygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b;}
};
class CRectangle: public CPolygon {
public:
int area ()
{ return (width * height); }
};
class CTriangle: public CPolygon {
public:
int area ()
{ return (width * height / 2); }
};
```

```
int main () {
CRectangle rect;
CTriangle trgl;
rect.set_values (4,5);
trgl.set_values (4,5);
cout<< rect.area() << endl;
cout<< trgl.area() << endl;
return 0;
}
```

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are: width, height and set_values().

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used protected access instead of private.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width // protected access
CRectangle::width // protected access
CPolygon::set_values() // public access
CRectangle::set_values() // public access
```

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

137

class CRectangle: public CPolygon { ... }

This public keyword after the colon (:) denotes the maximum access level for all the members inherited from the class that follows it (in this case CPolygon). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

class daughter: protected mother;

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default

constructor (i.e., its constructor with no parameters) and its destructor are always called when a new

object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class: derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

For example:

```
// constructors and derived classes
#include <iostream>
using namespace std;
class mother {
public:
mother ()
{ cout<< "mother: no parameters\n"; }
mother (int a)
{ cout<< "mother: int parameter\n"; }
};
class daughter : public mother {
public:
daughter (int a)
{ cout<< "daughter: int parameter\n\n"; }
};
class son : public mother {
public:
son (int a) : mother (a)
{ cout<< "son: int parameter\n\n"; }
};
int main () {
daughter cynthia (0);
son daniel(0);
return 0;
}
```

```
mother: no parameters
```

```
daughter: int parameter
mother: int parameter
son: int parameter
```

Notice the difference between which mother's constructor is called when a new daughter object is
created and which when it is a son object. The difference is because the constructor declaration of
daughter and son:

```
daughter (int a) // nothing specified: call default
son (int a) : mother (a) // constructor specified: call this
```

Multiple inheritance

139

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could
write:

```
class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;
```

here is the complete example:

```
// multiple inheritance
#include <iostream>
using namespace std;
class CPolygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b;}
};
```

```
class COutput {
public:
void output (int i);
};
void COutput::output (int i) {
cout<< i << endl;
}
class CRectangle: public CPolygon, public COutput {
public:
int area ()
{ return (width * height); }
};
class CTriangle: public CPolygon, public COutput {
public:
int area ()
{ return (width * height / 2); }
};
int main () {
CRectangle rect;
CTriangle trgl;
rect.set_values (4,5);
trgl.set_values (4,5);
rect.output (rect.area());
trgl.output (trgl.area());
return 0;
}
```

7.2.Polymorphism

Before getting into this section, it is recommended that you have a proper understanding of **pointers**and class inheritance.

Pointers

Variables and Memory

When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there.

When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the &and `*` operators:

1. `&x` evaluates to the address of `x` in memory.
2. `*( &x )` takes the address of `x` and dereferences it – it retrieves the value at that location in memory. `*( &x )` thus evaluates to the same thing as `x`.

Memory addresses, or pointers, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. Just a taste of what we'll be able to do with pointers:

• More flexible pass-by-reference

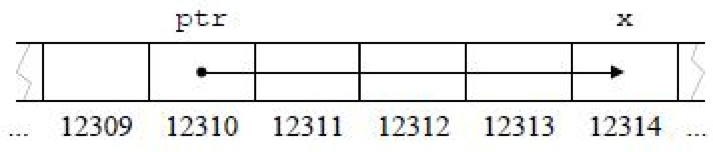
• Manipulate complex data structures efficiently, even if their data is scattered in different memory locations

Pointers and their Behavior

The Nature of Pointers

Pointersarejustvariablesstoringintegers – butthoseintegershappentobememory addresses, usually addresses of other variables. A pointer that stores the address of some variable `x` is said to point to `x`. We can access the value of `x` by dereferencing thepointer.

As with arrays, it is often helpful to visualize pointers by using a row of adjacent cells to represent memory locations, as below. Each cell represents 1 block of memory. The dot-arrow notation indicates that `ptr` "pointsto" `x` – that is, the value stored in `ptr` is12314, x's memory address.



Declaring Pointers

To declare a pointer variable named `ptr` that points to an integer variable named `x`:`int *ptr = &x;`

`int *ptr` declares the pointer to an integer value, which we are initializing to the addressof `x`.We can have pointers to values of any type. The general scheme for declaring pointers is:

```
data_type *pointer_name; // Add "= initial_value " if applicable
```

`pointer name` is then a variable of type `data type *` – a "pointer to a `data type` value."

Using Pointer Values

Once a pointer is declared, we can dereference it with the `*` operator to access its value:

```
     cout<< *ptr; // Prints the value pointed to by ptr,// which in
     the above example would be x’s value
```

We can use deferenced pointers as l-values:

```
     *ptr = 5; // Sets the value of x
```

Without the `*` operator, the identifier `x` refers to the pointer itself, not the value it points to:

```
cout<< ptr; // Outputs the memory address of x in base 16
```

Just like any other data type, we can pass pointers as arguments to functions. The same way we'd say void func(int x) {...}, we can say void func(int *x){...}. Here is an example of using pointers to square a number in a similar fashion to pass-by-reference:

```
void squareByPtr (int *numPtr){
```

Statement: Explained in:

int a::b(c) {}; Classes

a->b Data Structures

class a: public b; Friendship and inheritance

Pointers to base class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a

pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and

versatile feature, that brings Object Oriented Methodologies to its full potential.

We are going to start by rewriting our program about the rectangle and the triangle of the previous

section taking into consideration this pointer compatibility property:

// pointers to base class

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

```
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;


   // call triangle area.
   shape->area();


   return 0;
}
```

```
#include <iostream>
using namespace std;
class CPolygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
};
class CRectangle: public CPolygon {
public:
int area ()
{ return (width * height); }
};
class CTriangle: public CPolygon {
public:
int area ()
{ return (width * height / 2); }
};
int main () {
CRectangle rect;
CTriangle trgl;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
cout<< rect.area() << endl;
cout<< trgl.area() << endl;
return 0;
}
```

In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects
of classes derived from CPolygon, both are valid assignations.
The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the
area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.
In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it
in the base class. This is when virtual members become handy:
Virtual members
142
A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:

```
// virtual members
#include <iostream>
using namespace std;
class CPolygon {
```

```
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
virtual int area ()
{ return (0); }
};
class CRectangle: public CPolygon {
public:
int area ()
{ return (width * height); }
};
class CTriangle: public CPolygon {
public:
int area ()
{ return (width * height / 2); }
};
int main () {
CRectangle rect;
CTriangle trgl;
CPolygon poly;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
CPolygon * ppoly3 = &poly;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly3->set_values (4,5);
cout<< ppoly1->area() << endl;
cout<< ppoly2->area() << endl;
cout<< ppoly3->area() << endl;
```

```
return 0;
}
```
Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword
from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding
area() function for each object (CRectangle::area(), CTriangle::area() and
CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the
type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in
the above example.

A class that declares or inherits a virtual function is called a *polymorphic class*.

Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to
call its own area() function, which always returns 0.

Abstract base classes

Abstract base classes are something very similar to our CPolygon class of our previous example. The
only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an
abstract base classes we could leave that area() member function without implementation at all.
This is done by appending =0 (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:

```
// abstract class CPolygon
class CPolygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
```

144

```
virtual int area () =0;
};
```

Notice how we appended =0 to virtual int area () instead of specifying an implementation for the function. This type of function is called a *pure virtual function*, and all classes that contain at least
one pure virtual function are *abstract base classes*.

The main difference between an abstract base class and a regular polymorphic class is that because in
abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.

But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take
advantage of all its polymorphic abilities. Therefore a declaration like:

```
CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an
object. Nevertheless, the following pointers:

```
CPolygon * ppoly1;
CPolygon * ppoly2;
```

would be perfectly valid.

This is so for as long as CPolygon includes a pure virtual function and therefore it's an abstract base class. However, pointers to this abstract base class can be used to point to objects of derived classes.

Here you have the complete example:

```
// abstract base class
#include <iostream>
using namespace std;
class CPolygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
20
10
145
virtual int area (void) =0;
};
class CRectangle: public CPolygon {
public:
int area (void)
{ return (width * height); }
};
class CTriangle: public CPolygon {
public:
int area (void)
{ return (width * height / 2); }
};
int main () {
CRectangle rect;
CTriangle trgl;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
cout<< ppoly1->area() << endl;
cout<< ppoly2->area() << endl;
return 0;
}
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we
can create a function member of the abstract base class CPolygon that is able to print on screen the
result of the area() function even though CPolygon itself has no implementation for this function:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;
class CPolygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
virtual int area (void) =0;
void printarea (void)
{ cout<< this->area() << endl; }
};
class CRectangle: public CPolygon {
public:
int area (void)
{ return (width * height); }
};
class CTriangle: public CPolygon {
public:
int area (void)
{ return (width * height / 2); }
};
int main () {
CRectangle rect;
CTriangle trgl;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly1->printarea();
ppoly2->printarea();
return 0;
}
```

Virtual members and abstract classes grant C++ the polymorphic characteristics that make objectoriented
programming such a useful instrument in big projects. Of course, we have seen very simple
uses of these features, but these features can be applied to arrays of objects or dynamically allocated
objects.

Let's end with the same example again, but this time with objects that are dynamically allocated:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;
class CPolygon {
protected:
int width, height;
public:
```

```
void set_values (int a, int b)
{ width=a; height=b; }
virtual int area (void) =0;
void printarea (void)
{ cout<< this->area() << endl; }
};
class CRectangle: public CPolygon {
public:
int area (void)
{ return (width * height); }
};
class CTriangle: public CPolygon {
public:
int area (void)
{ return (width * height / 2); }
};
int main () {
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly1->printarea();
ppoly2->printarea();
delete ppoly1;
delete ppoly2;
return 0;
}
```

Notice that the ppoly pointers:

```
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
```

are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.

https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

https://beginnersbook.com/2017/08/cpp-function-overloading/