TYBSc Semester VI

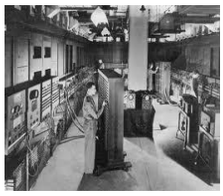
Applied Component
Electronic Instrumentation

Unit 4
C++ Programming

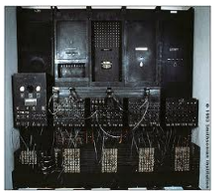
2016-2017

Associate Professor Pratibha P. Pai

Department of Physics
SIES College of Arts, Science & Commerce
Sion(west)



First generation
1940-1956



Second generation
1956-1963

Third generation
1964-1971

Fourth generation
1971- Present

Fifth generation
Present & Beyond

INTRODUCTION TO C++

Introduction to programming:- A computer is an electronic machine specifically designed to follow instructions. It can do a wide variety of tasks because it can be programmed. But it can do the only jobs that the programs tell it to do.

Programs and programming languages

A computer program is a set of instructions that tells the computer how to perform a task. A set of well defined steps for performing a task is called as an algorithm and the steps are sequentially ordered.

A programming language is a special language used to write computer programs. There are 2 categories of programming languages. They are -- low level and high level.

Low level language ---

- Is machine dependent and so not portable
- Difficult to understand and remember
- resembles the numeric machine language of the computer
- Uses mnemonics
- eg. Machine language (0s and 1s) and assembly language (mnemonics)

High level language---

- machine independent and portable
- Easy to understand and remember
- close to human language
- Uses English language
- eg. FORTRAN (FORmula TRANslation), BASIC(Beginners All-purpose Symbolic Instruction Code with various versions like GWBASIC, Q-BASIC, M-BASIC), COBOL(COmmon Business Oriented Language), C, C++(read as C plus plus), C#(read as C sharp), Unix, Java, SQL(Structured Query Language), HTML(Hyper Text Markup Language) etc

What is C++?

C++ is an Object-Oriented Programming (OOP) language. It was developed by Bjarne Stroustrup at AT and T Bell laboratories in Murray Hill, USA in 1985.

C++ is an extension of C developed by Dennis Ritchie. It was originally called as “C with classes” and later, the name was changed to C++.

Procedure-Oriented Programming(POP) and Object- Oriented Programming(OOP)

Conventional programming, using high level languages such as FORTRAN, COBOL and C, is commonly known as Procedure-Oriented Programming (POP).

In the Procedure-Oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. The technique of hierarchical decomposition is used to specify the tasks to be completed for solving a problem.

POP basically consists of writing a list of instructions for the computer to follow and organizing these instructions into groups known as functions. We normally use a flow chart to organize these instructions and represent the flow control from one action to another.

Following are some of the characteristic features exhibited by POP:

1. Emphasis is on doing things.
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data.
4. Data move openly around the system from system to system.

5. Functions transform data from one form to another.
6. Employs top-down approach in program design.

In the Object -Oriented approach, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications. So, Object- Oriented Programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

Basic concepts of Object- Oriented Programming

Some of the concepts are

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Brief note on basic concepts of OOP:

1. **Objects:** ***They are the basic run-time entities in an object-oriented system.*** They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user- defined data. Programming problem is analyzed in terms of objects and nature of communication between them.
   Program objects should be chosen in such a way that they match closely with real-world objects. Objects take up space in memory and have an associated address.

   When a program is executed, the objects interact by sending messages to one another. Each object contains data, and code to manipulate the objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message and the type of response returned by the objects.

2. **Classes:** ***A class is a collection of objects of similar type***. For eg: mango, apple and grapes are members of the class fruit. Classes are user – defined data types and behave like built – in types of a programming language. The entire set of data and code of an object can be made a user defined data type with a help of a *class*. Objects are variable of the type *class*. Once a class is defined, we can create any no. of objects belonging to that class. The syntax for defining a class is – fruit mango; where mango is an object belonging to class fruit.

3. **Data abstraction and encapsulation:** ***The wrapping up of data and functions into a single unit (called class) is known as encapsulation***. The most striking feature of a class is *encapsulation*. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. ***The insulation of the data from direct access by the program is called data hiding.***

   Data abstraction refers to putting together essential features without including the background details. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called the data members because they hold information. The functions that operate on these data are sometimes called member functions.

4. **Inheritance:** ***It is a process by which objects of one class acquire the properties of objects of another class.*** It supports the concept of hierarchical classification.

   The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

5. **Polymorphism:** ***The ability to take more than one form is called as polymorphism***.

   An operation may exhibit different behaviours in different instances. The behaviour depends on the types of data used in the operation. e.g For 2 numbers, the addition operator (+) gives the sum while for 2 strings, it gives another string(concatenation).The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.

   Polymorphism allows us to have more than one function with the same name in a program.

6. **Dynamic binding:** It is also known as late binding. Binding refers to the linking of a procedure call to the code to be executed in response to the call. ***Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time***. It is associated with polymorphism and inheritance.

7. **Message passing:** An object oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information in the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems. ***Message passing involves specifying the name of the object, the name of the function(message) and the information to be sent.***

Parts of a C++ program:- A sample program in C++ is as follows: -

```
// A program written for explanation, semester 6 of 2015-2016
 #include<iostream.h>
#include<conio.h>
void main (void)
 {
    clrscr( );
     int i;            //variable declaration statement
    cin>>i;        //input statement
    cout<< " The value of  i  is   "<< i<< endl;      //endl means end of line
   getch( );
}
```

Note the following (i) There are no line numbers.(ii) C++ is case-sensitive. (iii) C++ is space ignorant. (iv) All key words are in lower case.(v) Identifiers cannot have spaces in between. (vi) .cpp files are compiled. (vii) Header files are included. (viii) Multiple files need to be put in a project(.cpp files). ix) conio stands for console i/p- o/p. x) iostream stands for i/p –o/p stream.

Special characters:- Following are the special characters in C++.

< > → opening and closing brackets. # → pound sign(hash)
{ } → opening and closing braces. " "→ opening & closing quotation marks.
( ) → opening and closing parentheses. // → double slash
; → semi colon, the statement terminator.

The first statement in the program written above is a comment statement.

- A comment always starts with a double slash and terminates at the end of a line i.e it is a single line statement.

- There is no closing symbol in a comment.
- A comment may start anywhere in the line and whatever follows till the end of the line is considered as comment.
- Comments are not executed. The computer just ignores these.
- **So, we can write anything: name, date, reference, meaning of variable introduced etc...**
- Comments are only for the programmer to read and are used for documentation. They improve the readability.
- **Help other people to read & understand programs.**
- The // is a single line comment and so if the comment goes to the next line, // is used again.
- /* …… */ can also be used as in C. But here, whatever is written between /* and */ is the comment.(It may be a single line or it may be a number of lines.)

The second statement starts with # and it is called a preprocessor directive.

- ✓ The preprocessor reads the program before it is compiled and executes those lines beginning with a # symbol.
- ✓ It instructs the compiler to include the contents of the header file iostream into the sample file to facilitate i/p and o/p operations.
- ✓ The header file iostream should be included in the beginning of all programs that use input and output statements. Any program that outputs data to the screen or inputs data from keyboard must include this file.
- ✓ cin and cout are two predefined objects that represent standard i/p and o/p streams. standard i/p stream represents the key board while the standard o/p stream represents the monitor.

The third statement is again to include header file conio.

- It contains the instructions necessary for clearing the monitor for the output and to view the o/p on the monitor. (conio → console i/p - o/p header file)
  If you do not include this, then

(1) the o/p from the present program merges with the o/p from the previous program.
(2) When you run the program, the monitor gets cleared before you view the o/p of your program.
void main (void):- Every program must have a function called 'main'.
A function has 2 channels of communication (1) for receiving the information and (2) for sending the information. When the word 'void' appears before a function name and inside the function's parentheses, it means the function will not be sending information to or receiving information from any other function.
Whatever is written in between { and } is the content of the function 'main'. It is sometimes known as the body of the program.

Note that –

- All complete statements end with a statement terminator (;).
- << is an insertion operator, also called 'put to' operator.
- >> is an extraction operator, also called 'get from' operator.
- Multiple use of << or >> in one statement is called cascading.
- cin is used to read a number, a character or a string of characters from the standard input device ie. keyboard.
- cout is used to display an object onto the standard device ie. Video screen.
- clrscr(); implies clearing the screen to display the output from the program.
- getch(); is needed to view the output from the program on the screen.

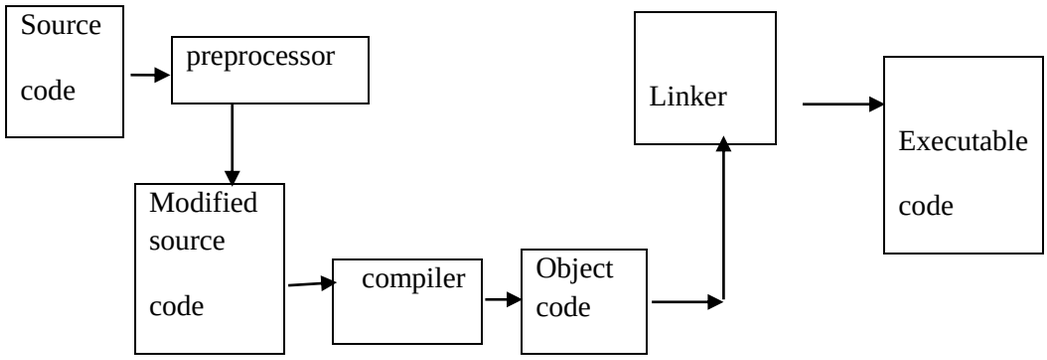
Source code, Object code and Executable code

A program written in C++ must be typed into the computer and saved in a file.

The statements written by the programmer are called source code and the file they are saved in is called the source file.

After the source code is saved to a file, it is translated to machine language. During the process of translation, a program called the preprocessor reads the source code. It searches for lines (starting with #) that contain commands which cause the preprocessor to modify the source code.

Now, the compiler steps through the preprocessed source code, translating each source code instruction into the appropriate machine language instruction. This process finds any syntax error in the program. (Syntax errors are illegal uses of key words, operators, punctuation and other language elements.) If no errors are found, the compiler stores the translated machine language instructions (called object code) in a file called object file.



During the last phase of translation process, another program called linker combines the object file with library routines. The library contains hardware - specific code for displaying messages on the screen and reading the input from the keyboard. Thus, an executable file containing machine language instructions (or executable code) is created and it is ready to run on the computer.

Many development systems, particularly those on personal computers, have integrated development environments (IDE). 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.

TOKENS:- The smallest individual units in a program are known as tokens. They are

- **key words**
- **identifiers (programmer- defined symbols)**
- **Operators**
- **constants and**
- **strings.**

C++ program is written using these tokens, white spaces and the syntax of the language.

KEY WORDS:- These are also known as reserved words. They have a special meaning. They cannot be used for any other purpose than what they are designed for. All the key words are written in lower case in C++.

Following is the list of key words in alphabetical order □

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, export, false, float, for, friend, goto, if, inline, int, long, mutable,

namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while.

- IDENTIFIERS(Programmer- defined symbols):- These are words or names of variables, functions, arrays, classes etc. defined by the programmer. They are the fundamental requirement of any language. Each language has its own rule for naming the identifiers.

Following are the rules in C++:

- A valid identifier is a sequence of one or more letters, digits or underline symbols (underscore _).
- The first character must be one of the letters a to z, A to Z, or an underscore character but not a digit.
- Neither blank spaces nor marked letters can be a part of an identifier. Special characters are also not allowed….e.g #, $, &, !, @, –, /.
- The length of an identifier is not limited, although for some compilers only the first 32 characters of an identifier are significant.
- Upper case and lower case characters are distinct. i.e count and COUNT are not same.
- A declared key word cannot be used as a variable name.

e.g. MyNumber, Aug2006, dayOfWeek, min_balance, accountNo, _perimeter_rect etc. are valid identifiers.

Some of the invalid identifiers are 2names, amt$, bank a/c, S.I.E.S, cin, int, a-z, 1 947 , 9,000 etc.

- **OPERATORS**:- An operator is a symbol that tells the computer to perform specific mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. These perform operations on one or more operands. An operand is usually a piece of data, like a number. e.g + is an addition operator, * is a multiplication operator, % is a modulo operator etc. C++ is rich in built-in operators.

  Following are the categories of operators
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Bitwise operators
  - Others

Arithmetic operators The arithmetic operators are used for manipulating numeric values and performing arithmetic operations. Fundamental arithmetic operators are +, -, * , / and %.

The symbols for the arithmetic operators are + : addition, - : subtraction,
* : multiplication, / : division(This gives the quotient) and % : modulo (This gives the remainder)

Operation	Operator	Algebraic expression	C++ expression	Result if a= 14, b= 6, c=4
Addition	+	b + 7	b + 7	13
Subtraction	-	b – c	b – c	2
Multiplication	*	ab	a*b	84
Division	/	$\frac{a}{c}$	a/c	3
Remainder	%	a mod c	a %c	2

Note: 1. % operator requires that **both operands be integers** and second operand be non-zero.

2. For modulo operation the sign of the result is always the sign of the first operand.

e.g. -14 % 3 = -2. -14 % -3 = -2. 14 % -3 = 2

3. Result of division operation is truncated quotient when both operands are integer quantities.
4. / operator denotes integer division if both arguments are integers & floating point division otherwise.
5. There is **no exponentiation operator in C++**.Raising a number to power requires the use of library function pow. eg. pow(4,2) => 4^2 =16. pow (a,0.5) => $a^{1/2} = \sqrt{a}$.

Depending on the **number of operands employed**, there are 3 types unary, binary and ternary. (operand means the data on which an operation is performed.

Unary operator requires one operand. e.g. negation sign (operator) -10, -a etc.(Equivalent to multiplying the value of a by -1)

The increment and decrement operators:

The increment operator is ++ and decrement operator is -- . Both are unary operators. These can be used either in **post fix mode or pre fix mode**.

Note: 1. In the post fix mode, the operator is placed after the variable(e.g i++; num -- etc.)
Postfix evaluates to the old value of the variable first.
2. In the pre fix mode, it is placed before the variable name(e.g ++i; -- num etc.).
Prefix does the addition/subtraction first.

To increment means to increment the value by 1 and to decrement means to decrease it by 1.

```
e.g  n =2;                                  n = 3;
     cout<< n++<<endl;                      cout<< ++n;
     cout<<n ;
```

Here use the variable first and then increment. So the o/p is 2
3

Here increment first and display. So, the o/p is 4

These operators are very helpful in the execution of loops both in mathematical expressions as well as in relational expressions.

To understand the functioning of these operators consider the following-

```
(i)  x=2;                  (ii)  x=2;                        (iii) x=2;
     y=x++;                      y=5;                              y=5;
     cout<<  x<<endl;            z= x * y++;                       z= x * ++ y;
     cout<<y;                    cout<<x<< ”  “ <<y<<”  “ <<z;     cout<<x<<y<<z;
```

o/p 3 2

o/p 2 6 10

o/p 2612

Binary operator requires two operands. e.g. assignment operator(=), +, -, *, /, %; total=princi + interest; a=b + c; a= b*c etc.

Ternary operator requires three operands. There is only one ternary operator in C++ namely ? :

The general format is: expression 1 ? expression 2 : expression 3;

e.g 1. x=5 ? y=2 : y=0; The meaning is: Is x=5?, if yes, then y=2 or else y=0.

e.g 2. x < 0 ? y = 10 : z = 5 ; or (x< 0) ? (y = 10) : (z=5); The meaning is : if (x < 0) then y= 10; else z = 5;

e.g 3. a = x > 0 ? 1 : 0 ; The meaning : if (x > 0) then a= 1 ; else a = 0;

Relational operators: Relational operators allow the user to compare numeric values and determine if one is greater than, less than, equal to or not equal to another. They are as follows : Relational operators can be used to compare only numbers but not strings.

meaning	Operator	Meaning	Operator
greater than	>	less than or equal to	<=
less than	<	equal to	==
greater than or equal to	>=	not equal to	!=

'= =' is an equal to operator (x==5) while ' = ' is an assignment operator.(x = 5)

All the relational operators are binary operators i.e they use 2 operands. e.g x < 5 is a relational expression. All expressions have a value and their value can be only true or false.

Assume x = 10 and y = 2

x > y is true ; x < y is false ; x > = y is true ; x < = y is false ; x != y is true

Logical operators : These operators connect two or more relational expressions into one, or reverse the logic of an expression. There are 3 logical operators &&, || and ! Their effect is as below-

operator	meaning	Effect
&&	logical AND	Connects 2 expressions into one. Both expressions must be true.
||	logical OR	Connects 2 expressions into one. One or both expressions must be true for the overall expression to be true.
!	logical NOT	Reverses the "truth" of an expression. It makes the true expression false and the false expression true.

The general format is

(1) expression 1 **&&** expression 2 (2) expression 1 || expression 2 (3) **!** (expression)

In the first case,

- expression1 is evaluated.(left operand)
- If expression1 is **false ,** expression2 is not evaluated.
- If expression1 is true , the value of the second expression is the final value.

In the second case,

- expression 1 is evaluated.
- If expression 1 is **true ,** expression 2 is not evaluated.
- If expression 1 is false , the value of the second expression is the final value.

In the third case,

- expression is evaluated.
- If expression is **true , ! expression** is false and If expression is **false, ! expression** is true

Assignment operators

Assignment operator assigns or copies a value into a variable or the result of an expression.

When value is assigned to a variable as a part of variable declaration, it is called initialization. The symbol = is called as assignment operator. **Syntax:** variable = expression

- Expression may be a single constant or complex combination of variables , operators & constants.

e.g int x = 12; // literal assignment, literal means constant
int y = x + 5; // assignment with an expression (including a literal)
int z = x * y; // assignment with an expression
x = y means the value of y is assigned to x leaving y unchanged.

Special assignment expressions:

They are of 3 types i) compound assignment ii) Embedded assignment iii) chained assignment.

* Compound assignment :
General format is : variable1 ***operator*** = variable2.

operator	How to use	Meaning
+=	a +=6;	a = a + 6;
-=	b -=7;	b = b – 7;
*=	c *= 9;	c = c * 9;
/=	d /=10;	d= d /10;
%=	e %= 50;	e= e % 50;

Let the variable in the statement be *count* .
count = count + 1 ≡ count + = 1; and count = count -1 ≡ count - = 1;
count = count % 5 ≡ count % = 5; etc.

* Embedded assignment : x= (y = 10) +10; ≡ y=10; and x=y + 10;
e.g. i = j = k = 0 ; is i = (j = (k = 0)); x = y = 3 ; is (x = (y = 3));
i = j += k ; is i = (j += k) ; i += j = k ; is i += (j = k);

* Chained assignment : x= (y =54); OR x = y = 54;
Note: A chained statement cannot be used to initialize variable at the time of declaration.
i.e int a = b =c =d = 1 ; is illegal

Bitwise operators

Bitwise operators are used to perform logical operations on the individual bits of integer values. They are basically used for testing or shifting bits left or right.
X<<3 // shift three bit position to left
b>>1 // shift one bit position to right
shift operators are used for multiplication and division by powers of two.

operator	Name	Function
&	bit and	Performs a logical AND on each bit of two operands.
|	bit or	Performs a logical OR on each bit of two operands.
^	bit xor	Performs a logical XOR on each bit of two operands.
~	complement	A unary operator complements each bit, known as bitwise negation operator.

<<	SHL	Shift Left
>>	SHR	Shift Right

Note: 1. For masking any bit, use bitand operator. e .g
110_{10} =0110 1110_2 and 2_{10} = 0000 0010_2 .
110 & 2 gives 0000 0010_2. All bits in 110 are masked but one bit.
2. For turning bits ON, use bitor operator.
3. For toggling bits, use bitxor.
4. For testing the value of an individual bit, use the bitand operator.(same as masking.)
5. bitwise left shift(or right shift): suppose seat =4 then seat<<2 gives 16.
(4 =0000 0100 , so shifting 2 positions will give 00010000 i.e 16 or 4^2)

Others

operator	Name	Meaning
::	Scope resolution operator	Used to uncover a hidden variable.(or allows access to the global version of a variable)
new	Memory allocation operator	Used to create an object
delete	Memory release operator	Used to destroy an object
endl	Line feed operator	Same effect as " \n", used to format data display
sizeof( )	sizeof operator	Used to determine the size of a data type or any variable on any system. The operator returns the number of bytes used by that item.(The data type or the variable is placed within the parentheses.)
setw	Field width operator	setw(n) specifies a field width n for printing the value of the variable. This value is right justified within the field.

Hierarchy of operators

Arithmetic operators		Relational operators		Logical operators	
1) unary negation				1) !	NOT
2)*, /, %	left to right	1) <, <=, >, >=	left to right	2) &&	AND
3) +, -	left to right	2) = =, !=	left to right	3) ||	OR

- **CONSTANTS:** Constants are fixed values that do not change during the execution of the program. Constants are also called literals. These include integers, characters, floating point numbers and strings. Literal constants do not have memory locations.
  - Integer constants
  - ✓ decimal integer → 3456
  - ✓ floating point integer → 21.6
  - ✓ octal integer → 025
  - ✓ hex integer → 0x68
  - Floating point constants which are extremely large or extremely small are written with E or e. e.g 6.59E 10, 5.893 e -10, 6.62 E -34 etc. (e10 means 10^{10}).
  - Character constants are enclosed in single quotation marks.

```
e.g     {   char key;
            key = 'M';                          O/P is
            cout <<key << endl;                 M
            key = 'n' ;
            cout << key << endl;                n
            }
```

In the output, the quotation marks are not displayed.

```
e.g     {  int x = 2,  y = -6 ;
          y =  5 * x ;                          O/P is
          x =  3 * y ;                          30
          cout <<  x << endl << y << endl ;     10
        }
```

- String constants are enclosed within double quotation marks.
  e.g " radius", " Cpp" etc.

ESCAPE SEQUENCE:

There are many escape sequences in C++. They give us the ability to exercise greater control over the way information is output by the program. An escape sequence always starts with a back slash (\) and is followed by one or more control characters.
(Note: There is no space between the back slash and the control character.)
Some of the common escape sequences are as shown in the table.

Escape sequence	Name	Description
\n	**New line**	Causes the cursor to go to next line for printing.
\t	**Horizontal tab**	Causes the cursor to go to the next tab stop.
\a	**Alarm**	Causes the computer to beep.
\b	**Backspace**	Causes the cursor to move left by one position.
\\	**Backslash**	Causes the backslash to be printed.
\'	**Single quote**	Causes the single quotation mark to be printed.
\"	**Double quote**	Causes the double quotation mark to be printed.
\r	**return**	Causes the cursor to go to the beginning of the current line.

Lines and statements:- A "line" is a single line as it appears in the program. It can be a blank line also. Blank lines make the program more readable. So in long programs, programmers always add as many blank lines as possible.

A statement is a complete instruction that causes the computer to perform some action. A statement can be a combination of key words, operators and programmer- defined symbols. Statements often occupy one line but sometimes they occupy more than one line.

Variables:-A variable is a named storage location in the computer's memory for holding a piece of information. The information stored in variables(actually stored in RAM) may change while running the program.

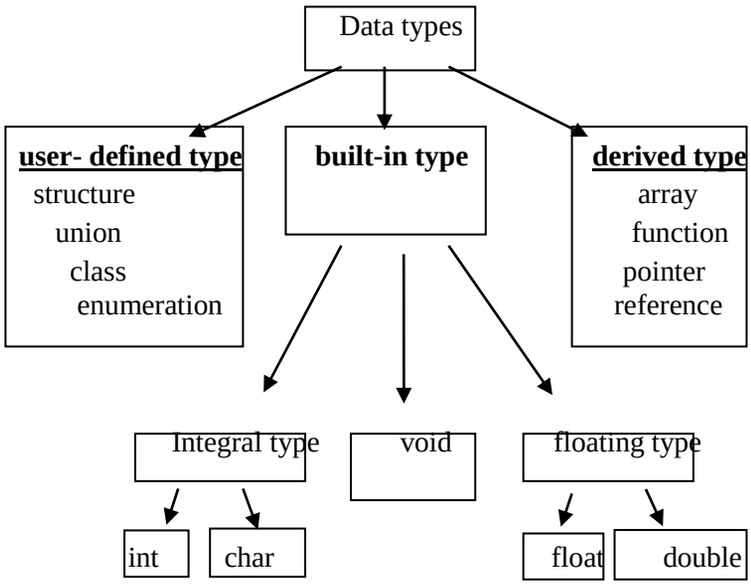
Input:- It is the information a program collects from the outside world.

Output:- It is the information that a program sends to the outside world.

DATA TYPES

Computer programs collect data from the real world and manipulate them in various ways. There are different types of data – whole numbers or fractional numbers, positive numbers or negative numbers, numbers very large or very small or textual information etc.

Data types can be classified under various categories as shown below:-



Variables are classified according to the data types which determine the kind of information that may be stored in them and the size of the variable. The size of a variable is the number of bytes of memory the variable uses.

integers(int) character(char) floating point numbers(float)

The words in parentheses above are key words.

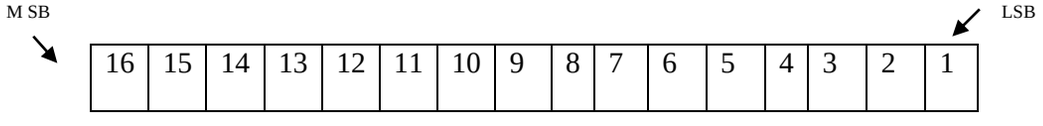
- Integer type data is further categorized as below –

The size of the integers is dependent on the computer. But generally, if int is 4 bytes, then short int is 2 bytes and long int is 4 bytes. In other words – short is no longer than int and long is no shorter than int.

Unsigned data types can store only positive values. They can be used only when you know that the program will not encounter any negative values.

For signed int, the 16^{th} bit (i.e MSB) is the sign bit. So signed int can have range

(-32,768 to +32,767 i.e 2^{15}).For unsigned int, the range is 0 to 65,535 i.e 2^{16}



We have the following:

- **int are at least as big as short int**
- **long int are at least as big as int**
- **unsigned short int are the same size as short int (2 byte)**
- **unsigned int are same size as int**
- **unsigned long int are the same size as long int (4 byte**)

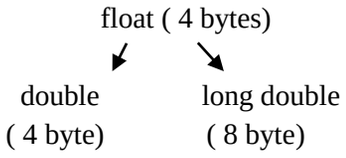
e.g of variable declaration – int i, int days, unsigned speed, short years, unsigned age, signed amount ….

- Floating point data type

These are used to declare variables that can hold real numbers. Internally, floating point

numbers are stored in a manner identical to scientific notation.

E.g $4.567 \times 10^4 = 4.567E4$,(45,670); $4.56 \times 10^{-4} = 4.56E\ -4$ (0.000456)



float data type is considered single precision while double is considered with double precision.

- **A double is at least as big as float and a long double is at least as big as double**.

float - numbers between ± 3.4 E ± 38 (i.e computer remembers 38 digits)
double - numbers between ± 1.7 E ± 308) i.e computer remembers 308 digits
long double - numbers between ± 1.7E ± 308)

On any computer, the above 3 types can be positive or negative.

- The char data type:- This is used for storing characters. (Actually, it is an int data type stored using ASCII). When a character is stored in memory, it is actually, the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.
  - A char requires one byte of memory.

e.g If A is to be the O/P , then the code segment will be either of the following.

```
{       char letter ;
    letter  =  65;   //( ASCII code for 65 is A )
        cout  << letter << endl;
     }
```

```
{     char letter;
        letter = 'A' ;  //character constant
        cout  << letter << endl;
   }
```

The bool data type:
Expressions that have a true or false value are called Boolean expressions.

The bool data type allows us to create small integer variables that are suitable for holding true or false values.

```
e.g  { bool    boolValue;
          boolValue  =  true ;                //   boolValue is the name of the variable,
           cout << boolValue << endl;                  no space in  between]
     }
```

O /P 1 True is represented in memory as 1 and false as 0.

Variables:

A variable is a named storage location in the computer's memory for holding a piece of information. The information stored in variables may change while the program is running. When information is stored in a variable, it is actually stored in RAM. It is obvious that the value stored in the variable changes during the program execution.

When the program is written, usually the first line in the function main is the variable declaration statement. It tells the compiler the variable's name and the type of data it holds. E.g. int count; or

int r ; or float radius ; char letter etc.(int stands for integer, count is the name of the variable. Similarly r , radius, letter are the names of the variables with r integer type, radius float type and letter character type.)These variables can be initialized also.

String comparison : For this, use #include< string.h > and the strcmp library function.

Its format is : strcmp(string 1, string 2) ;

The function compares the contents of string 1 with that of string 2 and returns one of the following values:

- If the 2 strings are identical, strcmp returns 0.
- If string 1 < string 2, strcmp returns a negative number.
- If string 1 > string 2, strcmp returns a positive number.

Actually, strcmp compares the ASCII codes of each character in the 2 strings. If it goes all the way through both strings finding no characters different, it returns 0. As soon as it finds two corresponding characters that have different codes, it stops the comparison. If the ASCII code for the character in string 2 is higher than the code in string 1, it returns a negative number and similarly positive number otherwise . Also strcmp is case sensitive.(Ram and ram are not same.)

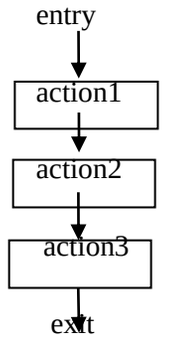
* **Advantage of strcmp is that names can be sorted alphabetically in ascending or in descending order.**

Structured Programming : Control structures

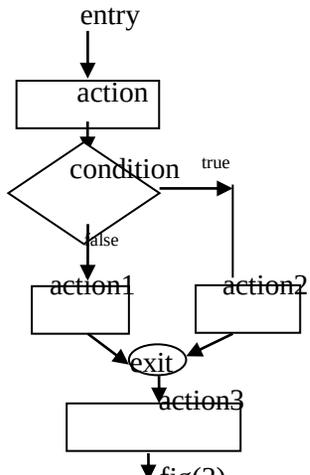
In C++ there are 3 control structures and all program processing can be coded by using only these 3 logic structures. They are

- Sequence structure(straight line)
- Selection structure(branching)
- Loop structure(iteration or repetition)

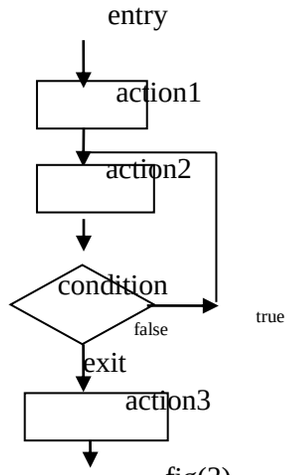
These structures are implemented using one-entry, one -exit concept. The approach of using one or more of the above structures is known as structured programming. Pictorially, they can be understood as shown below:



fig(1)
a linear sequence.



fig(2)
branching(if-then-else type).



fig(3)
loop structure(repeat until type).

Understanding: A program is usually not limited to a linear sequence of instructions.

During its process, it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what and how to perform the programs.

A block of instructions is a group of instructions separated by semicolon but grouped in a block delimited by the braces { }.

If the statement is a single instruction, then braces need not be used while if the statement is more than one instruction, then necessarily { } are to be used.

Following are the control – flow statements –

- if
- if – else } 2 way branch
- if – else if
- switch } multiple branch

  } Selection

- for
- while } entry -control
- do – while } exit -control

  } Loop

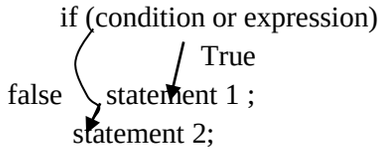
➧ **The *if* statement**:-

This statement is used to execute an instruction or a block of instructions **only if** a certain condition is fulfilled. The condition is the expression that is being evaluated. If the condition is true, the statement 1 is executed. If it is false, statement 1 is ignored and the program continues to execute statement 2.

The general format is

```
if (condition or expression)

    statement 1 ;
statement 2;
```



In the 'if' line, after the closing parenthesis, if semicolon is placed, then the statement following the 'if' line gets executed even if the condition is not true.

Note: You may write 'action' in place of 'statement' in the format above.

Example: **// check whether X is bigger than Y**

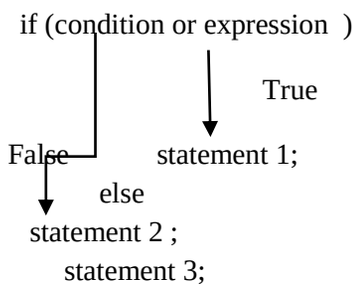
```
#include<iostream.h>
#include<conio.h>
void main( )
{clrscr( );
int X,Y;
cout<<" Enter the numbers";
cin>>X>>Y;
if(X>Y)
cout<<"The number "<<X<<" is bigger than"<<Y<<endl;
cout<<"The entered numbers are  "<<X<<" and  "<<Y.
getch( );
}
```

➧ **The *if – else* statement:**

This statement is an expansion of the *if* statement. The *if – else* statement will execute one group of statements if the expression is true and if false, another group of statements is executed.

The general format is

```
if (condition or expression)
   statement 1 or block 1 ;
else
   statement 2 or  block 2 ;
     statement 3;
```



Note: You may write 'action' in place of 'statement' in the format above.

Example: **Program to check whether the entered integer is even or odd**

The *if – else if* statement:

This is a chain of *if* statements, normally referred as *nested if- else* statement. They perform their tests one after another until one of them is found true. The format is

```
if (expression)
     statement 1 ;
  else if (expression)
     statement 2 ;
          // you can put many else if
      else if (expression)
         statement 3 ;
```

Here, the *else* part of one statement is linked to the *if* part of another. When put together, the chain of *if – else* statements becomes one long statement.

- **The *switch* statement**:-

This statement lets the value of a variable or an expression determine where the program will branch. It actually tests the value of an integer expression and then uses that value to determine which set of statements to branch to. The general format is –

```
switch(variable or expression)
 {
  case constant1:
     block of instructions 1;
     break;
  case constant 2:
     block of  instructions 2  ;
     break;
   default:
   block of instructions;
 }
```

First, the expression is evaluated. If it is constant1 then block of instructions 1 is executed until the '*break*' word is encountered. If it is constant2 then block of instructions 2 is executed until the '*break*' word is encountered and the process continues.. If the value of the expression is not any of the above mentioned constants, block of instructions under *default* is executed. Finally the control passes to the end of switch selective structure.

- For integer constants, use → case 1:, case 2:, case 3: …etc.
- For character constants, use → case 'A':, case 'B':, case 'C': …etc.
- You may write 'action' in place of 'block of instructions' in the format above.

- **The *while* loop**:- This is an *entry - controlled* loop structure.

The general format is as follows.

```
while (condition or expression)
  {   statement 1;
          OR
   Block of statements;
   }
     statement 2;
```

```
while (condition or expression)
                     True
      { statement 1 /Block of statements;
      }
False
      statement 2;
```

The *while* loop has 2 important parts.(1)An expression that is tested for a true or a false value and (2) a statement or a block of statements that is repeated as long as the expression is true. If false, the control exits the loop. So the entry is controlled.

It is to be noted that if there is only one statement following the *while* statement, one may use the braces or one may not.

Note: You may write 'action' in place of 'statement' in the format above.

The *do –while loop*:- This is an *exit - controlled* loop structure.

The general format is

```
        do
           {
           Statement 1;
             }
     while(condition or expression);
      statement 2;
```

```
        do
        {     Statement 1;
          }
   True
     while(condition or expression);        ; is a must.
                  False
              statement 2;
```

First, the Statement 1 under *do* gets executed. Then condition under *while* is tested.

If the condition is true, Statement 1 is executed again. It tests the expression again and Statement 1 is executed as long as the condition is true. So Statement 1 is executed after each iteration is complete.

If the condition is false, statement 2 gets executed.

If the condition in *while* is not true from the beginning itself, as the condition is checked later Statement 1 is executed at least once. So the exit is controlled.

The *for* loop:- This is an *entry-entrolled* loop.

The general format is

```
for( initial value ; test ;increment)
   { statements1 ;
   }
 statement2 ;
```

OR

```
for( initial value ;condition; post instruction)
   { statements1 ;
   }
 statement2 ;
```

The stepwise working is as follows: Step 1 → Step 2 —True→ Step 3(a)→ Step 4→ Step 5 (→ back to Step 2)

Step 2 —False→ Step 3(b)

```
    Step 1            Step 2                          Step 5
for (   initial value ;     test/condition ;     increment/decrement      )
          {          if true Step 3(a)
        statements1 ;         Step 4
      }
               statement2 ;
```

If false out to Step 3(b)

There are 3 parts inside the parenthesis separated by semicolon.

- The first part is the initializer or a starting value. This step is done only once. If the data type of this value is not declared earlier, it may be declared and used here.
- The 2nd part is the test expression. If it is true, the statements1 is executed. If false, statement2 is executed.
- The 3rd part is the increment/decrement or the update expression. It is executed at the end of each iteration.

The *for* – loop is a pre- test loop. It evaluates the test expression before each iteration. So if the initial, the final and the mean difference values are known then *for*- loop is the best loop to work with.

```
e.g.  for( int n=0 ; n <=6; n +=2)
        cout<< (n + n) << endl;
```

o/p

```
0
4
8
12
```

Some programming tasks require a running total to be kept. A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator.

Sentinel:

A sentinel is a special value that marks the end of a list of values. It signals that there are no more values to be entered. When the user enters the sentinel, the loop terminates.

```
e.g:    {     int count=0, point=0, total=0;
        cout<<"Enter no. of points earned & then -1 when finished \n";
         while(point !=  -1)
        {
             count++;
           cout<<"enter point for game :"<<count;
           cin>>point;
           if(point != -1)
           total+ = point;
        }
    cout<<"total points are="<<total<<endl;
   }
```

The *break* statement :This causes a loop to terminate early. When 'break' is encountered, the loop stops and the program jumps to the statement immediately following the loop:

```
e.g. int i=0;
      while(i++<10)
        {
          cout<<i<<endl;
          if(i = = 5)  // program stops after 5th iteration
          break;
         }
```

The *continue* statement: This causes the loop to stop its current iteration and begin the next one.

Comparison between *while* and *for* loop

'*while*'	*'for'*
• Initialization of the variable is outside and before the '*while*'	• Initialization of the variable is within the statement

statement • Incrementing the variable is inside the block of statements to be executed	• Incrementing the variable is within the *for*- statement

Comparison between *while* and *do-while*

'*while*'	'*do-while'*
• entry-controlled loop • condition is checked and if true, the block of statements gets executed • the block of statements get executed only if the condition is true • is a pre- test loop. • does not get executed at all if the condition is not true.	• exit –controlled loop • the block of statements get executed before checking the condition • even if the condition in '*while'* part is false, the block of statements under '*do'* get executed . • is a post- test loop • is executed at least once

.

Similar comparisons may be done with other statements.

A simple program to list odd integers from 1 to 10 in *for*, *while* and *do- while* forms

'for'	'*while*'	'*do-while'*

#include<iostream.h>
#include<conio.h>
void main(void)
{ int N;
for(N=1;N<10;N+=2)
{
cout<< N<<endl;
}
 	
getch();
}
	#include<iostream.h>
	#include<conio.h>
	void main(void)
	{ int N =1;
	while(N<10)
	{
	cout<< N<<endl;
	N+=2;
	}
	getch();
	}
		#include<iostream.h>
		#include<conio.h>
		void main(void)
		{ int N =1;
		do
		{ cout<< N<<endl;
		N+=2;
		}
		while(N<10);
		getch();
		}

MORE ABOUT PROGRAMMING

Commonly used header files for our programs:

1. <iostream.h >→ contains function prototypes for the standard input and standard output functions.
2. <math.h >→ contains function prototypes for math library functions.
3. <iomanip.h >→ contains function prototypes for the stream manipulators that enable formatting of streams of data.
4. <string.h >→ contains function prototypes for c-style string processing functions.

Dynamic initialization of variables

We know that a variable must be initialized using a constant expression and the compiler would fix the initialization code at the time of compilation. However in C++, initialization of the variable can be done at run time. And this is referred to as Dynamic initialization.

Example: 1. float Percent; ≡ float Percent = total/n;
Percent = total/n;
2. int L, B, AREA; ≡ int L, B;
AREA = L *B; int AREA = L *B;

Declaration of variables

C++ allows the declaration of variables anywhere in the scope. This means that a variable can be declared at the place of its first use. The program becomes easier and this reduces the error that might be caused by having to scan back and forth in lengthy programs. Also the program becomes easy to understand as the variables are declared in the context of their use.

e.g: for(int n=1; n<10; n++) int hour = min +sec;

Manipulators

These are operators that are used to format the display of data. Commonly used manipulators are *endl, setw* and *setprecision.*

endl:

- The *endl* manipulator is used in an output statement.
- It causes a linefeed to be inserted.
- It has the same effect as using the newline character "\n".
- The numbers with *endl* are left justified.

setw(n):

- The *setw(n)* specifies a field width n for printing the value of the variable.(n is a positive integer.)
- The values are right justified within the field.
- Character strings also get printed right justified.
- When *setw(n)*is used in a program, the header file iomanip.h has to be included.
- If a = 123 , b = 6782 and c = 8 and if the required o/p is

	1	2	3
6	7	8	2
			8

then use cout<< setw(4)<<a <<"\n"<<setw(4)
<<b<<"\n"<<setw(4)<<c<<endl;

- cout<<setw(10)<<"PRATIBHA\n"<<setw(10)<<"ASHA"; will give the o/p

		P	R	A	T	I	B	H	A
						A	S	H	A

setprecision(n):

- This is used to set the number of digits to be displayed after the decimal point in a floating point number .
- *setprecision(n)* displays the output number upto n decimal places.(n is a positive integer.)
- When *setprecision (n)*is used in a program, the header file iomanip.h has to be included.
- e.g: If the value of the variable 'area' is to be displayed 2 places after the decimal point, then use : cout<<" the value is = "<<setprecision(2)<<area;

Type cast operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

The following 2 formats may be used.

* (type-name) expression ▪ e.g i /(float) j; ▪ avg = sum_total /(float) n;	* type-name (expression) ▪ e.g i/ float(j); ▪ avg = sum_total /float(n);

Enumerated data type

An enumerated data type is a user-defined data type.

- The enumerated data type provides a way for attaching names to numbers.
- The keyword is *enum.*
- It automatically enumerates a list of words by assigning them values 0,1,2,3 … .
- The syntax for *enum* statement is
  enum shape{circle, square, rectangle}*;* *enum* colour{red,green,blue,white}; *enum* position{right, left};
  enum Day{ sun, mon, tue, wed, thurs, fri, sat}; *enum* radix{bin=2, deci=10, oct=8,hex=10};
- *C++ permits the creation of enums* without tag names(known as anonymous *enums).*
  enum {circle, square, rectangle}*;* *enum* {red,green,blue,white};
- In C++, the tag names become new type names ie. We may use
  shape circle*;* where circle is of type *shape*
  colour screen*;* where screen is of type *colour etc.*
- e.g: *enum* month {Jan =1,Feb,Mar, April, May, Jun, Jul};
  enum resistor_code { black, brown,red,orange,yellow,green,blue,violet,grey,white};

By default, enumerators are assigned integer values starting with 0 for the first enumerator.

However we may override the default.

EXAMPLE:1	EXAMPLE:2	EXAMPLE:2

{
clrscr( );
enum colour { red, blue, black};
cout<<red+black;
//u will get o/p as 2
getch();
}
	{
	clrscr( );
	enum colour { red =1, blue, black};
	cout<<red+black;
	//u will get o/p as 4
	getch();
	}
		{
		clrscr( );
		enum colour { red, blue=2, black};
		cout<<red+black;
		//red =0,black=3,u will get o/p as 3
		getch();
		}

Reference variable

A reference variable provides an alternative name(alias) for a previously defined variable.

- Reference variables are declared like regular variables but with an ampersand(**&**) in front of the variable name. e.g void board(int & refVar). [Variable refVar is called "a reference to an int"].
- Its creation is as below:
  data-type **&** reference- name = variable- name;
  e.g: int & refVar, float & avg, float & percent etc.
- A reference variable must be initialized at the time of declaration.
- Initialization of a reference variable is different from assignment to it.
- reference- name and the variable- name can be used interchangeably to represent the variable .
- E.g *float* & Sum = total*;*
  If total =10.9; is declared first then cout<<Sum; and cout<<total will give the same output.
  If total is incremented /decremented/used in an expression then output with Sum and output with total will be the same.

Symbolic constants

There are two types of creating symbolic constants

- ⋆ Using the qualifier '*const'* and
  Any value declared as '*const'* cannot be modified by the program.
  '*const'* is to be initialized.
  '*const'* allows the creation of typed constants.
  The use is similar to the use of #define but in # define there is no type information.
  e.g: const float PI =3.14; const int L=3; #define pi 3.14 etc.
- ⋆ By defining a set of integer constants using '*enum*' keyword.
  e.g: enum { R1, R2, R3 }; ≡ const R1 = 0, const R2 = 1, const R3 = 2.
  (Discussed already earlier.)

Expressions and their types

An expression is a combination of operators, constantsand variables arranged as per the rules of the language. An expression may consist of one or more operands and zero or more operators to produce a value. There are seven types of expressions. They are

- ✓ Constant expressions – consist of only constant values e.g 54; 20 +65/5; 'c'
- ✓ Integral expressions – produce only integer results e.g a*b; x* 'y'
- ✓ Float expressions- produce floating point results e.g 15.75; m+n
- ✓ Pointer expressions- produce address values e.g &x; "xyz"; ptr
- ✓ Relational expressions- (also called Boolean expressions)produce *bool* type results which take value 'true' or 'false' e.g j >i ; a+b==c ; x<=45
- ✓ Logical expressions –combine 2 or more relational expressions and produce *bool* type results e.g a>x && b>x ; p ==5 || q==7
- ✓ Bitwise expressions-used to manipulate data at bit level and are basically used for testing or shifting bits. e.g x<<4 meaning shift 4 bit positions to left.

In a program, a combination of the above expressions(called as compound expressions) may also be used.

FUNCTIONS

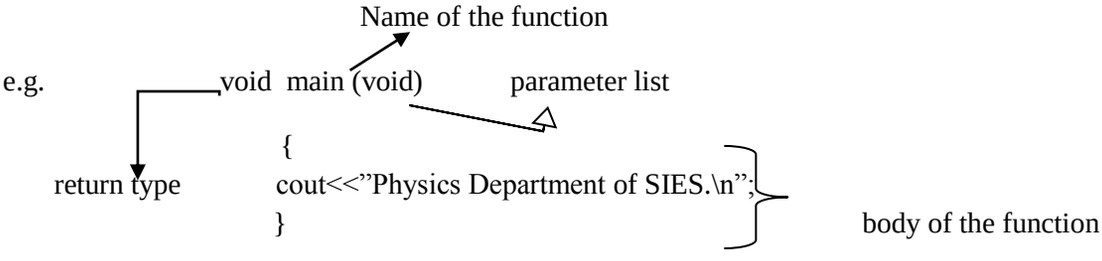
A function is a collection of statements that performs a specific task. Functions are building blocks of C++ programs. Two reasons for using functions are —

(i) A function breaks a program into small manageable units. Each unit is a module, programmed as a separate function similar to a book divided into different chapters or sections.

(ii) Functions simplify the programs and reduce the program size. If a specific task is repeated or performed in several places in a program, a function can be written once to perform that task and then be executed anytime it is needed. The function code is stored in only one place in the memory.

Defining and calling functions

A function definition contains the statements that make up the function. All function definitions have the following parts: a) Return type. b)Name. c) Parameter list. and d) body.



Return type:- A function can send a value to the part of the program that activated it. The return type is the data type of the value that is sent from the function. E.g. int, float, bool, void etc.

Name:- A function should have a name. Normally, rules applicable to variable names apply to function names. Eg . first, square, PayScale, MyAccount etc.

Parameter list:- The program can send information into a function. The parameter list is a list of variables that hold the values being passed to the function. E.g. (int), (void) etc.

(parameter list is a mechanism by which functions communicate with each other.)

Body:- It is the statements that perform the function's operation. They are enclosed in a pair of braces.

In the example **void main (void)** , 'void' is used both for return type as well as the parameter list of function 'main'. It means that the function does not receive any values when it starts and does not send a value when it finishes. This statement is called as function header.(Note no semicolon at the end.)

The functions that have e return value should use the '*return'* statement for termination.

void main (void) ≡ int main(void)

```
{       ------- ;
        ---------;
    return 0;
}
```

Calling a function

A function is executed when it is called. Function 'main' is called automatically when a program starts. But, all other functions must be executed by 'function call' statements. When a function is called, the computer temporarily puts the main function on hold and starts the execution of that function.

Function call or function invocation is the mechanism that transfers control to a function.

Example 1:

```
        // Program for understanding the working of a function.
  // There are 3 functions, main, one and two
   #include<iostream.h>
   #include <conio.h>
            void  one (void)     // function definition or function header, so no semicolon
              {
               cout<<" I am in  TYBSc\n";
               }

                void two (void)       // function header
                 {
                  cout<<" I am now in college.\n";
                    }

int main(void)
    {
      cout<<" I am in the beginning of function main.\n";
       one ( );          //call function one. Notice the semicolon
       two ( );          // call function two
       cout<<" Well, I have come back to main\n";
```

o/p
I am in the beginning of function *main*
I am in TYBSc
I am now in college.
Well, I have come back to *main*

```
 getch();
return 0;
}
```

Note:

- The function header is a part of the function definition. It defines the function's return type, name and parameter list.
- The function call is a statement that executes the function. So it is terminated with a semicolon like all C++ statements. The return type is not listed in the function call and if the program is not passing information into the function, the parentheses are left empty. Function call statements may be used in loops, *if*-statements and *switch* -statements.
- In a program, any number of functions and function calls may be present. One function *main* must necessarily be there in a C++ program.
- A function may be called within another function also.

Example 2 : // There are 3 functions *main, college* and *dept*

```
// function dept is within function college
#include<iostream.h>
#include<conio.h>
        void dept (void)            // function header
        {    cout<<" This is second branch\n";
        }

    void college (void)             // function header
    {   cout<<"This is first branch\n";
        dept ( );                   //call function dept
        cout<<"Back to function college \n";
    }

    void main (void)
{   cout<< "This is SIES college in function main\n";
    college ( );                    //call function college
  cout<<"Back to SIES college\n";
 getch();
}
```

o/p

```
This is SIES college in function main
This is first branch
This is second branch
Back to function college
Back to SIES college
```

Note: Some people use "*main*" as the last function while some use it in the beginning.

Function prototypes

Before a function is being called, the compiler should know the function's return type, number of parameters it uses and the type of each parameter. There are 2 ways of doing this.

- By using function header at the beginning of the program(as in Example 1 & 2) and
- By declaring the function with a function prototype (as in Example 3).

The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a template is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed and the return value is treated correctly.

Values that are sent into a function are called arguments.

e.g. void displayValue (int num)

The variable num is a parameter(or formal argument) . The above enables the function displayValue to accept an integer value as an argument.

The form of a function prototype is: **type function-name (argument-list);**
e.g 1. void *one* (void); // function header is similar but for a ;
The meaning : the function *one* has a void return type and uses no parameters.

e.g 2. float *area*(float x, float y);
The meaning : the function *area* has a float return type and uses 2 parameters which are of data type float.
e.g 3. float *volume*(float x, int y, float z);

e.g 4. float *volume*(float , int , float);
The meaning : the function *volume* has a float return type and uses 3 parameters x and z are float type while y is int type.
Note:

- ✓ If you do not use function header or function prototype ahead of all calls to the function, the program will not compile.
- ✓ It is necessary to declare independently all variable arguments inside the parantheses.
- ✓ It is necessary to indicate the data type for all arguments in the list. (see e.g 2/3above)
- ✓ It is optional to write the variable names(see e.g 4.above)as the compiler only checksfor the type of arguments.
- ✓ If variable names are used in the prototype, the same names need not be used in the function call or function definition. (see **Example** 4.below.)

Default arguments

In C++ a function can be called without specifying all the arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call.

- ✓ Default values are specified when the function is declared.(see e.g 1 below).
- ✓ Default values are specified similar to variable initialization. (see e.g 1 below).
- ✓ Only trailing arguments can have default values. (see e.g 2 below).
- ✓ Default values are added from right to left. (see e.g 2 below).
- ✓ They are useful when an argument has the same value throughout the program.
- ✓ The complier looks at the prototype to see how many arguments the function uses and if if found less, then default values are assigned to that argument.
- ✓ Even after mentioning the default value in the prototype, when the call to the function is made, the value to that variable may be passed explicitly. (see e.g 3 below).

e.g 1. int *total*(int theory, float prac, int proj=10); //prototype with default
total (49, 38.8) // call function *total*, default value not mentioned.
e.g 2. float *amount*(int princi, int year, int month=6,float rate=4.5*);* √
float *amount*(int princi, int year=3, int month, float rate=4.5*);* ×
e.g 3. float *product*(int a, int b, int c=6, float d= 3.14*);* √
product(4, 3, 5,*);* // call with c=5 and one argument missing √
e.g 3. float *amount*(int princi, int year, float rate=4*);* √
amount(2000,5,3*)* // No argument missing. √

Example 3 :

```
//program using function prototype.
            // add and subtract are 2 functions.
            #include<iostream.h>
            void add ( );    // function prototyping
            void subtract ( );
           int main (void)
           {    clrscr( );
            cout<<"Calling function add ( ).”<<endl;
            add ( );          // a call to function add
            subtract ( );    // a call to function subtract
            cout<<"       a       return       from       both
    functions”<<endl;
           getch( );
          return 0;
          }
       void  add ( )         //this is function header
       {
              int  i ,j, sum;
             cout<<"\nInput 2 integers :”<<endl;
            cin>>i>>j;
           sum=i+j;
          cout<<"The sum of 2 integers=”<<sum<<endl;
        }
      void subtract ( )           // function header
       {
       cout<<"The prototyping is clearly understood.”<<endl;
       }
```

o/p
Calling function *add* ( ).
Input 2 integers :4
5
The sum of 2 integers=9
The prototyping is clearly understood.
a return from both functions

Example 4 :

```
// A function with 3 parameters.
                #include <iostream.h>
               #include<conio.h>
                void showSum(int, int, int);  //prototype ,note  only data type
                  int main(void)
               {     clrscr();
                int  A1, A2, A3;
                cout<<"Enter 3 integers to find sum”;
              cin>>A1>>A2>>A3;
             showSum(A1,A2,A3);    // function call
            //call to function showSum with 3 arguments
           // see data type ‘int’ should not written in a call
               getch();
             return 0;
              }

void showSum(int num1,int num2, int num3)    // here int to be written –a must
        {
          cout<<(num1+num2+num3)<<endl;
        }
```

o/p
Enter 3 integers to find sum
3
4
5
12

Example 5 : //default value

```
#include<iostream.h>
#include<conio.h>
float amount(int , int , float r=0.5);
int main(void)
{ clrscr( );
int a;
cout<<"enter a"<<endl;
cin>>a;
cout<<a<<endl;
amount(100,2);
 getch( );
 return 0;
 }
 float amount(int p, int n, float r)
 {float amount;
  amount=p*n*r;
 cout<<"total amount="<<amount;
 }
```

o/p

```
enter a
5
5
total amount=100
```

Example 6 : //default value changed

```
#include<iostream.h>
#include<conio.h>
float amount(int , int , float r=0.5);
int main(void)
{ clrscr( );
int a;
cout<<"enter a"<<endl;
cin>>a;
cout<<a<<endl;
amount(100,2,0.1);
 getch( );
 return 0;
 }
 float amount(int p, int n, float r)
 {float amount;
  amount=p*n*r;
 cout<<"total amount="<<amount;
 }
```

o/p

```
enter a
2
2
total amount=20
```

Call by reference and Return by reference

We know that a reference variable provides an alternative name (alias)for a previously defined variable. In C++, provision of this reference variable is useful in passing parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.(Any changes made to the reference variable are actually performed on the variable for which it is an alias.)

Prototype for *board* function is void *board*(int &);

Note: & should appear both in prototype as well as in the header that uses a reference variable as a parameter but in function call, it should not appear. See **Example 7.**

If a function uses more than one reference variable as a parameter then before each reference variable name,& should be placed .eg. void *add* (int &, int &, int &);

Return by reference: A function can also return a reference.

Inline functions: An inline function is a function that is expanded in line when it is evoked. This means that the compiler replaces the function call with the corresponding function code.

The inline functions are defined as follows:

```
inline function-header
  { function body
    }
```

It is easy to make a function inline. Just prefix the keyword inline to the function definition. All inline functions must be defined before they are called. Usually the functions are made inline when they are small to be defined in one or two lines. See **Example 8.**

Example 8:

```
//use of inline function
#include<iostream.h>
#include<conio.h>
inline float mul(float x, float y)
{ return (x * y);
```

Example 7:

```
// use of ref. variable as a function parameter.
     //call by reference
     #include<iostream.h>
     void doubleNum(int &);        //prototype
     void main(void)
    {
   int value =3;
 cout<<"In main, the value is "<<value<<endl;
 cout<<"calling doubleNum  "<<endl;
doubleNum(value);        //function call
cout<<"Back into main, the value is"<<value<<endl;
      }
         void doubleNum(int &refVar)
            {
               refVar* =2;
            }
```

```
}
inline double div(double p,
double q)
{ return (p/q);

}
int main()
{clrscr();
float a=12.345, b=9.82;
cout<<mul(a,b)<<"\n";
cout<<div(a,b)<<"\n";
getch();
return 0;
}
```

o/p
121.22789
1.257128

Returning a value from a function

A function may send a value back to the part of the program that called the function. Although several arguments may be passed into a function, only one value may be returned from it. The data type of the return value precedes the function name in the header and the prototype.

E.g int square(int); This statement is a function prototype. Function name is square. It accepts an integer argument and returns an integer.

int square(int num) // function header can be written this way also.

{ return num*num; }

See Example 9.

Example 9:

```
//returning a value
# include<iostream.h>
 int square(int);   //prototype
 void main(void)
{
 int value, result;
  cout<<"Enter a number and" <<endl;
cout<<" then you see"<<endl;
cin>>value;
  result = square(value);   // call
cout<<value<<"squared is"<<result<<endl;
}
int square(int num)    // header
{
return num*num;
}
```

o/p
Enter a number and
then you see
10
10 squared is 100

The 'return' statement

When the last statement in a function has finished executing, the function terminates and the program returns to the statement following the function call. However, it is possible to force a function to return before the execution of the last statement using a return statement. When this statement is encountered, the function immediately terminates and the program returns.

Syntax : return;

Exit function

C++ program stops executing when the end of the function main is reached or when a return statement in main is encountered. The program does not stop when other functions end. Control of the program goes to the place immediately following the function call. In order to terminate the program in a function other than main , exit function is used . So, when the exit function is called , the program stops irrespective of the function containing the call.

See **Example 10.**

Note: (1) stdlib.h header file is to be included.

(2)The function takes an integer argument.

(3)There are 2 named constants EXIT_FAILURE and EXIT_SUCCESS defined in stdlib.h. The former is defined as the termination code that commonly represents an unsuccessful exit under the current operating system while the latter represents successful exit. One can use exit(0).

Example 10:

```
// use of exit(0)
 #include<iostream.h>
#include<conio.h>
#include<stdlib.h>
 void function(void);
 void main(void)
  {
   clrscr( );
   function( );
   getch( );
   }
 void function(void)
{  cout<<"This program";
 cout<<" terminates now."<<endl;
exit(0);
cout<<"What will you do now?"<<endl;
}
```

o/p:

```
This program terminates now.
```

Example 11:

```
// function over loading
 #include<iostream.h>
#include<conio.h>
 int volume(int);          //prototype 1
double volume(int, int);  //prototype 2
int  main ( )
   {   cout<<"function volume is overloaded.\n";
    cout<<volume(5);   //call 1
 cout<<"\n"<<volume(5, 2); //call 2
   getch( );
 return 0;
   }
//function definition for a cube of side a
int volume(int a)
{ return  a * a *a ;
}
// for a cuboid with square base
 double volume(int x, int y)
{ return  x * x * y ;
}
```

o/p:

```
function volume is overloaded.
125
50
```

Function overloading

'Overloading' refers to the use of the same thing for different purposes. C++ permits overloading of functions. This means that we can use the same function-name to create functions that perform a variety of different tasks. In Object Oriented Programming, this is referred to as ' function polymorphism'.

e.g : Name of the overloaded function is *add*

Sr. No.	Prototype	Function call
1	int *add(int a, int b);*	*add*(3,6);
2	int *add(int a, int b, int c);*	*add*(3,6,1);
3	double *add(int a, float b, float c);*	*add(4, 6.9, 2.25);*

The function 'add' would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function types. See **Example 11.**

A function first matches the prototype having the same number and type of arguments (best match) and then calls the appropriate function for execution.

Following steps are involved in the function selection:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotion to the actual arguments, such as *char* to *int* and *float* to *double* to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If there are multiple matches, error message is generated.
4. If all the above steps fail, then the compiler will try the user- defined conversions(used in handling class objects)in combination with integral promotions and built-in conversions to find a unique match.

****

Some questions in UNIT 4

1. What are the logical operators used in C++? Write a program to support your answer.
2. Write note on different data types.
3. Write a note on various operators.
4. What are the Basic concepts of Object- Oriented Programming? Explain any 3 of them.
5. Write a note on enumerated data type.
6. Write a note on arithmetic and relational operators.
7. Write the structure of for-loop. Explain with the help of an example. How can you convert a 'for' loop into a do-while loop?
8. Compare the if-statement with if- else statement. Write a program and explain. What do you understand by 'nested if-else' ?
9. What is a switch statement? Write the general format and explain with a program to support.
10. What do you understand by 'break' and 'continue' statements in C++ programming? Explain with suitable examples.
11. Write a program in C++ to find the maximum of 3 positive integers entered through the key board.
12. Write a program in C++ to input a 3 digit number and find the sum of the digits in it.
13. Write a program in C++ to find the area of a circle using #define statement.
14. Explain with suitable examples the do- and do- while loops in C++.
15. Write a program to find whether the entered letter is a vowel or not. Use **switch** statement.
16. With the help of an example, write briefly on 'function header', 'function call' and 'function prototype'.
17. Explain with examples, the difference between local and global variables.
18. Differentiate between passing parameter by value and passing parameter by reference.
19. With suitable programs, explain functions with more than one variable. Write its output expected.
20. What is the use of setprecision (n) and setw (n)? When they are used in a c++ program, which header file is included? Explain with the help of an example.
21. Write a program in C++ to display all the odd numbers between 20 and 50 using for- as well as do- while loop.
22. Write a program in C++ to calculate the following sum and also print the sum …..
    (1 * 2) + (3 * 4) + (5 * 6) + (7 * 8) + (9 * 10)
23. Input principal amount, rate of interest and print the value of simple interest for a period of 5 years. The program should repeat as many times as the user wishes.---use do-while
24. Write a program in C++ to list 1 to 15 in a line vertically. ----use the for-loop
25. Write a program in C++ to input a temperature in Kelvin and convert it to Celsius.
26. Write a program in C++ to find the factorial of an integer input at the time of execution.
27. Write a program in C++ to find the sum of square s of integers 1 to 50 using the *while* loop.

==========