OBJECT ORIENTED DESIGN WITH C++
AN INTRODUCTION

Software is a collection of programs. ***Program*** is a set of statements that performs a specific task. Since the invention of the computer, many programming approaches have been tried. These included techniques such as ***Modular Programming, Top-Down Programming, Bottom-Up Programming and Structured Programming.***

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980's. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object Oriented Programming (OOP) is an approach to program organization and development that attempts to elimpinate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language.

PROCEDURE ORIENTED PROGRAMMING

The technique of hierarchical decomposition has been used to specify the tasks to be completed in order to solve a problem… Procedure oriented programming basically consist of writing a list of in structions for the computer to follow, and organizing these actions and represent the flow of control from one action to another, A list of instructions for the computer to follow, and organizing these instructions into groups known as ***functions.***

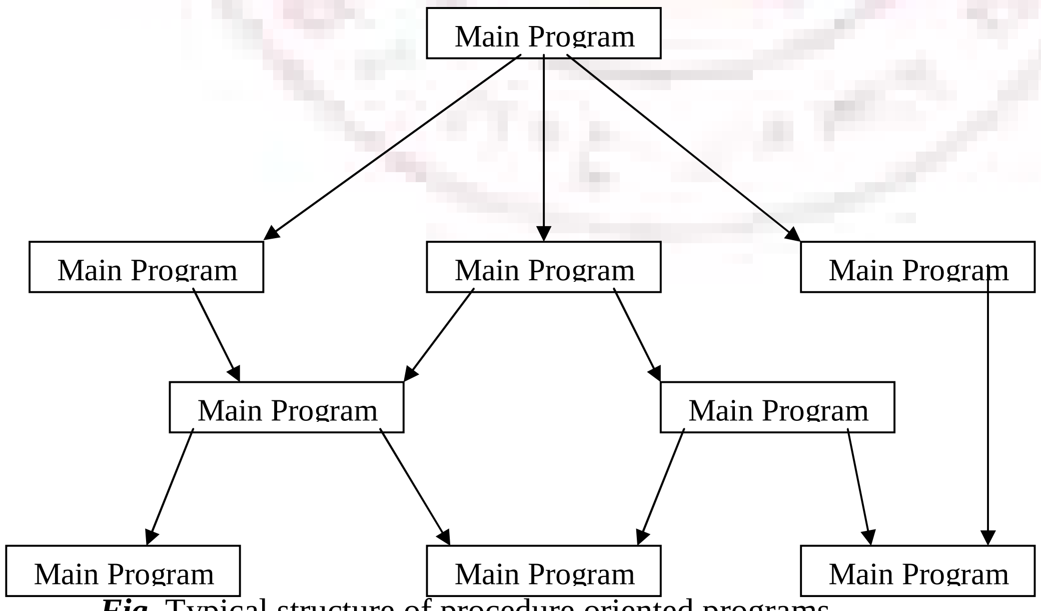
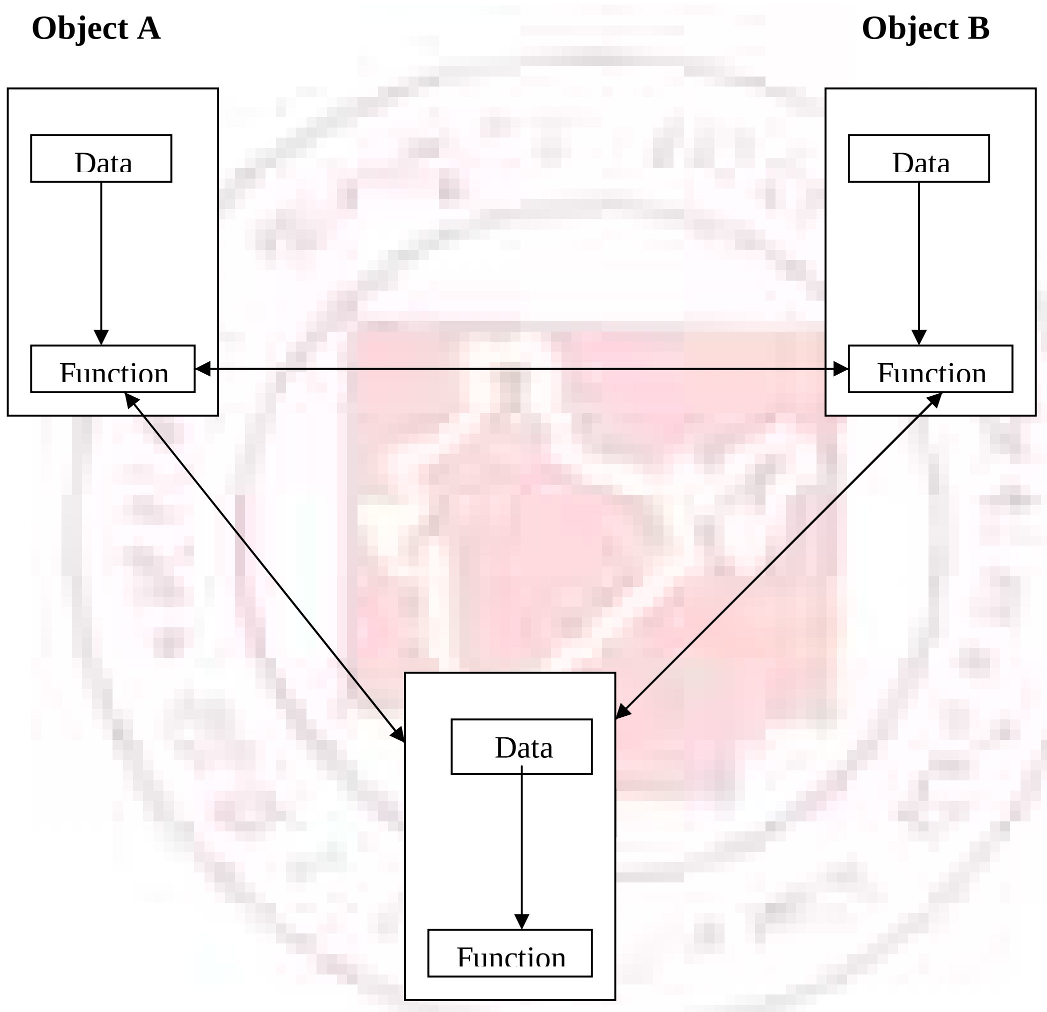


Fig. Typical structure of procedure oriented programs

OBJECT ORIENTED PROGRAMMING

Object Oriented Program (OOP) treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside function. OOP allows us to decompose a problem into a number of entities called object and then builds data and functions around these entities.



Object C
Fig. Organization of data and functions in OOP

Basic Concepts of Object Oriented Programming

1. Objects
2. Data Abstraction
3. Inheritance
4. Dynamic binding

5. Classes
6. Data Encapsulation
7. Polymorphism
8. Message Passing

The detailed information of the basic concepts of Object Oriented Programming will be discussed in later.

OBJECT ORIENTED LANGUAGES

The languages should support several of the OOP concepts to claim that they are object oriented. Depending upon the features they support, they can be classified into the following two categories.

1. Object based Programming languages
2. Object Oriented Programming languages

Object based programming is the style of programming that primarily supports encapsulation and object identity.

Major features that are required for object- based programming are

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator Overloading.

Languages that support programming with objects are said to be object based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features namely Inheritance and Dynamic binding.

Object-oriented features = Object-based features +Inheritance + Dynamic binding
Example: for Object-Oriented programming languages
C++, Smalltalk and Object Pascal.

C++ AN INTRODUCTION

C++ is an Object-Oriented Programming language. Initially named “C with Classes”, C++ was developed by Bjarne Stroustrup at AT &T Bell Laboratories in Murray Hill, New Jersy, USA in the early eighties. C++ is the superset of C. Most of the thinks are applied to C++.

Applications of C++

C++ is a flexible language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life application systems.

Since C++ allows us to create hierarchy-related objects, we can build special object-oriented libraries which can be used later by many programmers.

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.

The three most important facilities that C++ adds on to C are classes, function overloading, and operator overloading.

DATA TYPES

There are five atomic data types in C: character (char), integer (int), floating-point (float), double (double), and valueless (void). C++ adds two more bool and wchar_t. the exact format of floating point values will depend upon how they are implemented. Integers will generally correspond to the natural size of a word on the host computer. Values outside that range may be handled differently by different compilers.

The range of float and double will depend upon the method used to represent the floating minimum range for a floating point value is 1E-37 to 1E-37. The type void either explicitly declares a function as returning no value or creates generic pointers.

The following figure illustrates the size of the data type occupies in the memory

Data type	**Typical size in bits**	**Minimal Range**
char	8	-127 to 127
unsigned char	8	0 to 255
signed char	8	-127 to 127
int	16	-32,767 to 32767
unsigned int	16	0 to 65,535
signed int	16	Same as int
short int	16	-32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	Same as short int
long int	32	-2,147,483,647 2,147,483,647
signed short int	32	Same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

IDENTIFIERS AND KEY WORDS

IDENTIFIERS

Identifiers can be defined as the name of the variables and some other program element using the combination of the following characters.

Alphabets	**:**	**a..z, A...Z**
Numerical	**:**	**0..9**
Underscore	**:**	**_**

Special characters

All characters other than the above are treated as special characters. For example [], blank space, ( ) etc.

In C++ language, upper case and lower case letters are destined and hence there are 52 letters in all. A variable should not begin with a digit. C++ does not set a maximum length for an identifier.

The following are valid identifiers	***The following are invalid identifiers***
MyName	4ab
I	name ( )
I4	first name
h_name	

KEYWORDS

The keywords are also identifiers but cannot be user defined since they are reserved words. The following words are reserved for use as keywords. We should not choose them as variables or identifiers.

Asm	continue	float	new	signed	try
Auto	default	for	operator	sizeof	typedef
Break	delete	friend	private	static	union
Case	do	goto	protected	struct	unsigned
Catch	double	if	public	switch	virtual
Char	else	inline	register	template	void
Class	enum	int	return	this	volatile
Const	extern	long	short	throw	while

CONSTANTS

The value that cannot be changed during execution is called constant, ie. Unmodifiable

General syntax:

Const x=10; //Value could not changed during the execution.

There are three types of constants

- String Constants
- Numeric Constants
- Character Constants

STRING

A String constant is a sequence of alphanumeric characters enclosed in double quotation marks whose maximum length is 255 characters.
Example: "Hi Student"

NUMERIC

Numeric constants are positive or negative numbers. There are four types of numeric constants: Integer constant, floating point constant, hex constant and octal constant. An integer constant may either be of single precision or double precision.

Numeric consents	**Integer**	Integer
		Short Integer (short)
		Long Integer (long)
	Float	Single precision (float)
		Double precision (double)
		Long double
	Unsigned	Unsigned char
		Unsigned integer
		Unsigned short integer
		Unsigned long integer
	Hex	Short hexadecimal
		Long hexadecimal
	Octal	Short octal
		Long octal

Example: for Integer Constants

```
Const int x=100;
Const short int y=1;
Const long int z=1000123;
```

CHARACTER

A character represented within single quotes denotes a character constant.

Examples

‘A’
‘a’
‘:’
‘?’

BACKSLASH CONSTANTS

The backslash (\) is used to denote non-graphic characters and other special characters for a specific operation. The following characters are used as non as non-graphic characters”

‘\a’	alert a bell character	‘\n’	new line (line feed)
‘\t\	horizontal tab	‘\b’	back space
‘\r’	carriage return	‘\f’	form feed
‘\v’	vertical tab	‘\\’	backslash
‘\”	single quote	‘\”’	double quote
‘\0’	null character	‘\?’	question mark

Expressions

An Expression is a collection of data object and operands that can be evaluated to a single value. An object is a constant, variable or any other expression.

General Form	***Example***
Object1 operator Object2	2+3 evaluates 5
	6/2 evaluates 3

OPERATORS

Operators are used to evaluate the objects /variables.

ARITHMETIC OPERATORS

Arithmetic operations are the basic and common operations performed using any computer programming. If two variables (operands) are operated by an operator than the operators is normally called binary operator. If it is an single variable it is called unary operator

OPERATOR	**MEANING**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder of an integer division)

Example:

```
Int x=10’ y=50, Z;
Z=x + y;
```

ASSIGNMEMT OPERATORS

An assignment operator is used to assign back to a variable, a modified value of the present holding. The symbol = is used as an assignment operator and it is evaluated at the last.

OPERATOR	**MEANING**
=	Assign a value to a variable
+ =	Add and assign the value
- =	Subtract and assign the value
*=	Multiply and assign the value
/=	Divide and assign the value
%=	Remainder should be stored back to the variable
>> =	Right shift and assign (Shift operator discussed in later)
<< =	Left shift and assign (Shift operator discussed in later)
& =	Bitwise AND operation and assign(Bitwise operators discussed In later)
\ =	Bitwise OR operation and assign (Bitwise operators discussed In later)
~ =	Bitwise Complement and assign (Bitwise operators discussed In later)

Example

A=5;
Assign the value 5 is into the variable A
B=6
B*=5
B contains the value 30

COMPARISON OPERATORS

The comparison Operator can be grouped in to two categories: Relational Operator and Equality Operators.

OPERATOR	**MEANING**	**CATEGORY**
<	Lass than	Relational Operator
>	Greater than	Relational Operator
<=	Lass than or equal to	Relational Operator
>=	Greater than or equal to	Relational Operator
==	Equal	Equality Operator
!=	Not equal to	Equality Operator

General Syntax

Var1 <= var2

LOGICAL OPERATORS

OPERATOR	**MEANING**
&&	Logical AND
| |	Logical OR
!	Not

Logical AND

A compound expression is true when two conditions (expressions) are true. *The results of the Logical AND operator becomes.*

A	**B**	**Result A & B**
False	False	False
False	True	False
True	False	False
True	True	True

General Syntax:

(expression1 && expression2)

Logical OR

A compound expression is true either one condition is true. The result of the Logical OR operator becomes.

A	**B**	**Result A | | B**
False	False	False
False	True	True
True	False	True
True	True	True

General Syntax

(expression1 || expression2)

Logical NOT

A logical expression can be changed from false to true or from true to false with the negation operator.

The result of the logical Not operator becomes.

A	Result! A
True	False
False	True

General Syntax

(! Expression)

BITWISH OPERATORS (OR BITWISE LOGICAL OPERATORS)

OPERATOR	**MEANING**
&	Bitwise AND
^	Bitwise Exclusive OR
|	Bitwise (Inclusive) OR
>>	Bitwise right shift
<<	Bitwise left shift
~	Bitwise complement

SPECILE OPERATORS

A. Unary Operators

The unary operators require only a single expression operand to produce a line. Unary operators usually precede their single operands. Sometimes, some operators may be followed by the operands such as incrementer and decrementer.

OPERATOR	**MEANING**
*	Contents of the storage field to which a pointer is pointing
&	Address of a variable
-	Negative value
!	Negation (0 if value 1, 1 if value 0)
~	Bitwise complement
++	Increment
--	Decrement
type	Forced type of conversion
sizeof	Size of the subsequent data type or type in byte

b. Ternary Operator (? :)

The? : Conditional operator is ternary operator:

General Syntax:

Expression1? expression2:expression3

If Expression1 is True then expression2 is evaluated otherwise expression3 is evaluated.

Example:

X=6;
X<5? A=0; a=1
Now the a value is 1, because x<5 is false.

c. Comma Operator (,)

The comma separates the elements of a function argument list. It is also used in comma expressions.

Example:

Int x, y;
It is explained in detail in the topic control statements.

a) Scope Resolution Operator (::)
b) Pointer- to- member declarator (::*)
c) Pointer- to- member operator (->*)
d) Pointer- to- member operator (.*)
e) New and delete operators
f) Line feed operator end l
g) Field width operator set

TYPE CONVERSION

In certain situations, some variables are declared as integers but sometimes it may be required to get the result as floating point numbers. The type conversion is to convert the set of declared type of some other required type. It is easy to convert values from one type to another type.

Example:

Int x = 10;
Float y = 11.4;
X=y
The above statements are valid. After the execution of the statement x=y, x value is 11. ie the x value is truncated.

Conversions can be carried out in two ways,

- Converting by assignment
- Using Cast operator

a) Converting by assignment

It is usual way of converting a value from one data type to another by using the assignment operator (= operator)

Example:

Char ch ='a';
Int I=ch;
Currently the I value is 97 ie the ASCII value of a.

b) Cast operator

Converting by assignment operator is carried out automatically but one may not get the desired result. The cast operator is a technique to forcefully convert one data type to the other. The operator used to force this conversion is known as the cast operator and the process is known as casting.

General Syntax:

(cast-type) expression;
Or
Cast-type (expression);

Example:

Int x=6, y=5;
Float result;
Result= (float) x/y;
Result 1=x/y;
Result 1 contains the value 1 and result contains 1.2

DECLARATION OF VARIABLES

In C, all variables must be declared before defining any executable statement in a program, but C++ allows declaration of the variables before and after executable statements. A declaration is a process of naming of the variables and their corresponding data types in C++.

The variable must be declared by specifying the data type and the identifier.

General Syntax:

Data-type identifier1, id2, id3, …, idn;

Example:

Char ch;
Where ch is the character data type.
Short int x, y;
Where x, y are short integer data type and hold data size of 16 bits in length.
Long int x1;
Where x1 is a long integer data type whose maximum size is 32 bits in length.
Unsigned int limit;

Where limit is a variable and it has been declared as an unsigned integer data type.

STATEMENTS

A statement in a computer program carries out some action. There are thee types of statements used in C++. They are expression statement, compound statement, and control statement.

(i) Expression statement

An expression statement consists of any valid C++ expression followed by a semicolon. The expression statement is used to evaluate a group of expressions.

```
X=y;
Sum= x + y;
```

(ii) Compound statement

A group of valid C++ expressions placed within a {and} statement is called a compound statement or a block. The individual statement may be of an expression statement, a compound statement or even a control statement. Note that the compound statement is not completed with a semicolon.

Example:

```
{
A=b+c;
X=x*x;
Y=a+x;
}
```

(iii) Control Statement

The control statement is used for the program flow and to check the conditions of the given expression or a variable or a constant. The keywords of the control statements are normally a predefined or reserved word and the programmer may not use them as ordinary variables.

COMMENTS

Comments start with a double slash symbol and terminate at the end of line. A comment may start anywhere in the line whatever follows till the end of the line is ignored. The comment statement does not compile during the compilation.

Single line commenting

```
// Statement
```

Multi line commenting

```
/*
Statements
*/
```

SIMPLE C++ PROGRAMS

A typical C++ program would contain four
Sections: include files, class declaration, class functions and definitions and the main program

Include files
Class declarations
Class functions and definitions
Main function program

The ***include directories*** or (***Pre processor directives***) specifies the header file which contains the functions are used in our program.

Example:

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

It will be discussed later.

Class declarations, Class functions and definitions are specified after the header files included.
Finally the ***main function*** program. It is the entry point of our program.

IOSTREAM.H

The standard header file input and output stream <iostream.h> contains a set of small <u>and specific general purpose functions for handling input and output data.</u> The I/O stream is a sequence of following characters written for the screen display of read from the keyboard. The standard input and output operations in C++ are normally performed by using the I/O stream as cin for input and cout for output. C++ allows three types of stream classes namely.

Istream	consists of input function to read a stream of characters from the keyboard.
Ostream	consists of output functions to write a character onto the screen.
Iostream	supports both input /output stream of function to read a stream of characters from the keyboard and to display a stream of objects into the video screen.

INPUT AND OUTPUT STATTEMENTS

a) cout

The **cout** is used to display an object onto the standard device, normally the video screen. The insertion operator (<<) is used along with the cout stream.

General System:

```
Cout<<"String" <<var2<<var3<<…<<varn

Example:
Int x=100;
Char ch='a';
Cout<<x<< ch;
Cout<<"The output is "<<x;
```

b) cin

The cin is used read a number, a character or a string of characters from a standard input device, normally the keyboard. The extraction operator (>>) is used along with the cin operator.

General Syntax:

```
cin >> var1>>var2>>..>>varn;
```

Example:

```
Int a, b;
Cin>>a >> b;
```

A complete C++ program

Program 1: A simple c++ program that prints "This is may first program in C++"

```
#include<iostream.h>                        //Include the pre processor directives
Void main ()                                // Main function
{                                           // block open
Cout <<"This is my first program in C++";   //Statement
}                                           // block closed
```

Program 2: A program to read any two numbers through the keyboard and to perform simple arithmetic operations.

```
//Program for Addition of two numbers
#include <iostream.h>
Void main ()
{
Int a, b;
Cout<<"Enter the two numbers" <<endl;
Cin>>a>>b;
Cout <<”Addition value=”<<a +b <<endl;
Cout<< “Subtraction value=”<<a-b<<endl;
Cout <<”Multiplication value=”<<a*b<<endl;
Cout <<Division value=”<<a/b<<endl;
}
```

MANIPULATOR FUNCTIONS

Manipulator functions are special stream functions that change certain characteristics of the input and output. They change the format flags and values for a stream. The main advantage of using manipulator functions is that they facilitate the formatting of input and output streams.

The following are the list of standard manipulators used in a C++ program. To carry out the operations of these manipulator functions in a user program, the header file input and output manipulator **<iomanip.h>** must be included.

Predefined manipulators

Following are the standard manipulators normally used in the stream class.
Endl is an output manipulator to generate a carriage return or line feed character

Example:

```
Cout<<”Hello” <<endl;
```

Setbase() manipulator is used to convert the base of one numberic value into another base. In addition to the base conversion facilities such as to bases dec, hex and oct, the Setbase() manipulator is also used to define the base of the numeral value of a variable.

Hex hexadecimal base (base 16)
Dec decimal base (base 10)
Oct octal base (base 8)

Example:

```
#include<iostream.h>
#include <iomanip.h>
Void main()
{
Int value=10;
Cout <<”Decimal base=”<<dec<<value<<endl;
Cout<<”Hexa decimal base=”<<hex<<value<<endl;
Cout<<”Octal base=”<<oct<<value<<endl;
}
Or
#include<iostream.h>
#include<iomanip.h>
```

```
Void main()
{
Int value=10;
Cout<<"Decimal base="<<Setbase(10)<<value<<endl;
Cout<<"Hexa decimal base="<<Setbase(16)<<value<<endl;
Cout<<"Octal base="<<Setbase(8)<<value<<endl;
}
```

Output:

```
Decimal base=10
Hexa decimal base=a
Octal base=12
```

Setw() stands for set width. It is used to specify the minimum number of character positions on the output field of a variable.

General syntax:

Setw(int w)

The above statement changes the field width to w, but only for the next insertion. The default field width is 0.

Example:

```
Int a=10;
Cout<<Setw(1)<<a<<endl;
Cout<<Setw(10)<<a<<endl;
```

Output:

```
10
        10
```

Setfill() manipulator function is used to specify a different character to fill the unused filed width of the value.

General syntax:

Setfill(char ch)

The above statement changes the fill character to f. The default fill character is a space.

Example :

```
Int a =10;
Cout<,Setfill('*');
Cout<<setw(5)<<a;
```

Output:

```
***10
```

Setprecision() is used to control the number of digits of an output stream display of a floating point value.

General Syntax:

Setprecision (int p)

The above statement is used to set the precision for floating point insertions to p. The default precision is 6.

Example:

```
        Float c=5/3;
        Cout<<Setprecision(1)<<c<<endl;
        Cout<<Setprecision(2)<<c<<endl;
        Cout<<Setprecision(3)<<c<<endl;
        Output:
        1.7
        1.67
        1.667
```

Ends is a manipulator used to attach a null terminating character (‘\0’) at the end of a string

Example:

```
        Cout<<”Hello”<<ends<<”World”;
```

Output:

```
Hello World
```

Ws stand for white space. It is used to ignore the leading white space that precedes the first field.

Example:

```
        #include<iostream.h>
        #include<iomanip.h>
        Void main()
        {
        Char name[15];
        Cout<<”Enter a line of text\n”;
        Cin>ws;
        Cin>>name;
        Cout<<name<<endl;
        }
```

Output:

```
        Enter a line of text
        Hello world
        Hello
```

Flush is used to cause the stream associated with the output to be completely emptied.For output on the screen, this is not necessary as all output is flushed automatically. However, in the case of disk file being copied to another, it has to flush the output buffer prior to rewinding the output file for continued use.

Example:

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

```
{
Cout<<"Enter a line of text\n";
Cout<<"Hello world\n";
Cout.flush();
}
```

Output:

```
Enter a line of text
Hello world
```

Note: hex, dec,oct,ws,endl,ends and flush manipulators are defined in stream.h. The other manipulators are defined in iomanip Which must be included in any program.

Setiosflags manipulator function is used to control different input and output settings. The I/O stream maintains a collection of flag bits.

General Syntax:

Setiosflags(long int)

Resetiosflags performs the same function as that of the reset function. The flags representaed by the set bits in f are reset

General Syntax:

Resetiosflags(long 1)

Example:

```
#include<iostream.h>
#include<iomanip.h>
Void main()
{
Int value=10;
Cout<<Setiosflags(ios::showbase);
Cout<<Setiosflags(ios::dec);
Cout<<value<<endl;
Cout<<Setiosflags(ios::hex);
Cout<<value<<endl;
Cout<<Setiosflags(ios::oct);
Cout<<value<<endl;
}
```

Output:

```
10
Oxa
012
```

INPUT AND OUTPUT (I/O) STREAMFLAGS

To implement many of the above manipulators, I/O streams have a flag field that specifies the current settings.

The flag names and their meanings are given below.

FLAG NAME	**MEANING**
Skipws	Skip white space during input
Right	Left justification of output
Internal	Pad after sign or base indicator
Dec	Decimal base
Oct	Octal base
Hex	Hexa decimal base
Show base	Show base for octal and hexa decimal numbers
Show point	Show the decimal point for all floating point numbers
Uppercase	Shows uppercase hex numbers
Showpos	Show '+' to positive numbers
Scientific	Use E for floating notation
Fixed	Use floating notation
Unitbuf	Flush all stream after insertions
Stdio	Flush stdout, stderr after insertion

Arguments for setflags

FIRST ARUMENT (FLAG NAME)	**SECOND ARGUMENT**
Ios:: skipws	
Ios::left Ios::right Ios::internal	Ios::adjustfield
Ios::dec Ios::oct Ios::hex	Ios::basefield
Ios::showbase Ios::showspos Ios::uppercase	

Ios::showspoint	
Ios::scientific Ios::fixed	Ios::floatfiled
Ios::Unitbuf Ios::stdio	

(a) Turning the bit format flag on

In order to change the state of the cout object, the bits that represent its state must be changed. The setf () function is invoked for setting the bit format flags of the I/Ostream.

General Syntax:

Cout.setf (flags to be set);

The bitwise OR (|) operator is used in the argument list of the setf() function in order to change the bit format flag more than one.

Example:

Cout.setf (ios:: showbase|ios::showspoint|ios::uppercase);

(b) Turning the bit format flag off

The unsetf(() function is used to change the bits directly off. This function takes exactly one argument to turn off the bit pattern

General Syntax:

Cout.unsetf(flags to be turned off);

Example:

Cout.unsetf(Ios::uppercase);

The bitwise OR (|) operator is used in the argument list in order to turn off more than one bit format flag of the I/O stream.

(c) Base field bit format flag

The basefield format flag is used to display integers in the proper base

Ios::dec	shows integers in decimal format
Ios::oct	shows integers in octal format
Ios::hex	shows integer in hexadecimal format

Only one of the above can be set at any time. These format flags control the base in which numbers are displayed. By default, dec is set.

General Syntax:

Cout.setf(iosd::dec,ios::basefiels);
Cout.setf(ios::oct,ios::basefield);
Cout.setf(ios::hex,ios::basefield);

(d) Show base bit format flag

The showcase format flag is used to display the base for octal and hexadecimal numbers. If showcase is set, this flag prefaces integral insertions with the base indicators used with C++ constants. If hex is set, for instance, an 0X will be inserted in front of any integral insertion. This flag is not set by default. The unsetf() function is invoked to undo the base setting.

General Syntax:

Cout.setf(ios::showbase);

(e) Showpos bit format flag

The Showpos format flag is used to display the sign of an integer. If this flag is set, a "+" sign will be inserted before any integral insertion. It remains unset by default note that on positive decimal output, the '+' is assumed, but by default it will not appear. If the number is negative, the '-' sing will always appear. The unsetf() function is invoked undo the setting of a positive sign.

General Syntax:

Cout.setf(Ios::showspos);

(f) Uppercase bit format flag

The uppercase bit format flag is used to display output in uppercase. By default, the following notations always appear in lowercase.

- ❖ A hexadecimal number (a,b,c,d,e,f)
- ❖ The base of hexadecimal number (0x)
- ❖ A floating point number in the scientific notation (2.3E3)

The **unsetf()** function is used to revert back to lowercase.

General Syntax:

Cout.setf(Ios::uppercase);

(g) Formatting floating point numbers

The following sections explain how floating values are formatted using the different flag settings in C++.

(h) Show point bit format flag

The show point bit format flag is used to show the decimal point for all floating point values. By default, the number of decimal position is six.

General Syntax:

Cout.setf(ios::showpoint);

The **unsetf()** flag is invoked to undo the showing of all decimal values of a floating point number.

(i) Precision

The precision member function is used to display the floating point value as defined by the user.

General Syntax:

Cout.precision(int n)

Where n is the number of decimal places of the floating value to be displayed.

Example:

Cout.precision(5);

(j) Float field bit format flag

Sometimes a floating point value may have to be displayed in scientific notation rather than in fixed format.

- **Scientific**

When scientific is set, floating point values are inserted using scientific notation. There is only one digit before the decimal point followed by the specified number of precision digits which in turn is followed by an uppercase or a lowercase depending on the setting of uppercase and the exponent value.

General Syntax:

Cout.setf(ios::scientific,ios::adjustfield);

- **Fixed**

When fixed is set, the value is inserted using decimal notation with the specified number of precision digits following the decimal point. If neither scientific nor fixed is set (the default), scientific notation will be used when the exponent is less that or greater than precision. Other wise, fixed notation is used.

General Syntax:

Cout.setf (ios::fixed, Ios::adjustfield);

(k) Adjust field bit format flag

The adjust field consists of three field settings

Ios::left	left justification
Ios::right	right justification
Ios::internal	pad after sign or base indicator

- **Left**

Only one of these may be set at anytime. If left is set, the inserted data will be flush left in a field of characters width wide. The extra space, if any, will be filled by the fill character

General Syntax:

Cout.setf (Ios::left, Ios::adjustfield);

- **Right**

  If right is set, the inserted data will be flush right.

General Syntax:

Cout.setf (Ios::right, Ios::adjustfield);

- **Internal**

  If internal is set, the sign of a numeric value will be flush left while the numeric value flush right and the area between them will contain the pad character.

General Syntax:

Cout.setf (Ios::internal, Ios::adjustfield);

(l) Fill and width

If the total number of characters needed to display a field is less than the current field width, the extra output spaces will be filled with the current fill character. In C++, it is permitted to use any character to serve as the fill character. But by default it is blank.

- **Fill**

  The fill () function is used to specify a new fill character. Once it is specified, it remains as the fill character unless it is changed.

General Syntax:

Cout.fill (char ch);
Where ch is a character to be filled.

Example:

Cout.fill (‘*’);

- Width

  The width () function is used to specify the size of the data variable.

General Syntax:

Cout.width(int n);
Where n is a total field width of a variable.

Example:

Cout.width (10);

(m) Unitbuf

When Unitbuf is set, the stream is flushed after every insertion. This flag is unset by default

General Syntax:

Cout.setf (ios::unotbuf);

(n) Studio

This flag flushes the stdout and stderr devices defined in Stdio.h. This is unset by default.

General Syntax:

Cout.setf (ios::stdio);

(o) Skipws

If set, leading white space is ignored on extraction. By default skipws is set.

CONTROL STATEMENTS

Control statement can be classified into three types

1. conditional Statements
2. Iterative Statements
3. Breaking Statements

1. CONDITIONAL STATEMENTS

The conditional expressions are mainly used for decision making. In the subsequent sections, the various structures of the control statements and their expressions are explained. The following statements are used to perform the task of the conditional operations.

(i) If statement

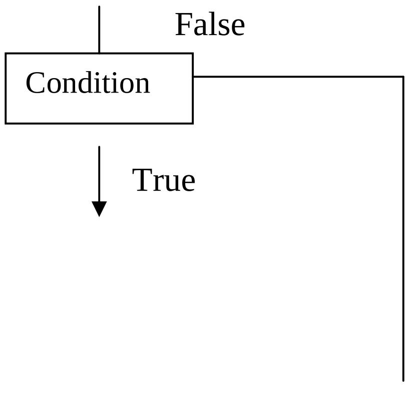
The if statement is used to express conditional expressions. If the given condition is true then it will execute the statements or block; otherwise the control goes to the next statement.

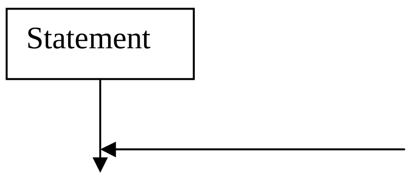
General Syntax:

```
If (expression)
{
        Statement1;
        Statement2;
        ................
        Statement;
}
```

If the expression is true then the statements within if block are executed.

Flow Diagram:





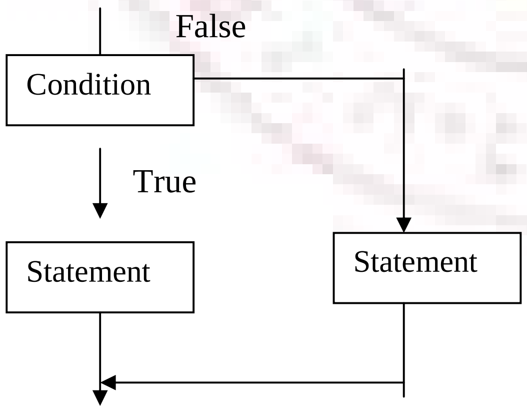
(ii) If-else statement

The if-else statements are used to express conditional expressions. If the given condition is true then it will execute a statements (s); otherwise another set of statements(s).

General Syntax:

```
If (expression)
{
        Statement s1;
        Statement s2;
        …………
        Statement sn;
}
Else
{
        Statement t1;
        Statement t2;
        ………...
        Statement tm;
}
```

<u>Flow Diagram</u>



If else lader

```
If (expression)
{
}
        Else if(expression)
        {

        }
        Else if(expression)
        {

}
Else
{
}
```

(iii) Switch-case statement

Switch Statement is a special multiway decision maker that tests whether an expression matches one of the number of constant values, and braces accordingly.

General Syntax:

```
        Switch (expression)
        {
        Case constant 1: statements;
        Break;
        Case constant2; statements;
        Break;
        ………………………….
        ………………………….
        Case constantn: statement;
        Break;
        Default:      statements;
        }
```

The expression whose value is being compared may be any valid expression including the value of the variable, an arithmetic expression, logical comparison, a bitwise expression or the return value from a function call, but not a floating point expression. The constants in each of the case statements must obviously be of the same match occurs, the statement following that is executed. The statement can be either a simple or a compound statement.

The value that follows the keyword case may only be constants; they cannot be expressions. They may be integers or characters, but not floating point numbers or character strings.

The last case of this statement which is called the default case is optional and should be used according to the program ‘s specific requirement. The default statements will be executed when the expression is not matched by any other cases.

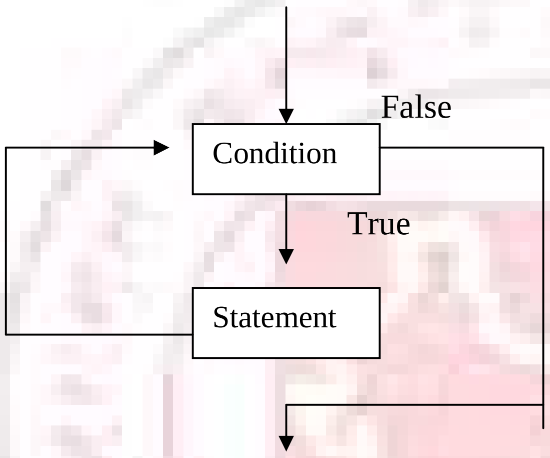
2. ITERATIVE STATEMENTS

Iteration is the process of repeating a statement or a group of statements a specified number of times. It is also called **Loop statements.**

The Iterative statements further classified into two types: **Pre-test Iteration, Post-test Iteration**

(i) Pre-test Iteration

A check before the iteration commences. Therefore there is possibility that the block is never executed if the condition is not valid for iteration at the very first encounter.



Example: for loop, while loop

For loop

The for loop is the most commonly used statement in C++. This loop consists of there expressions.

The first expression is used to initialize the index value, the second to check whether or not the loop is to be continued again and the third to change the index value for further iteration.

General Syntax:

```
For (expression_1; expression_2;expression_3)
{
      Statement 1;
      Statement 2;
      …………
      …………
      Statement n;
}
```

Where expression_1 is initial condition
expression_2 is test condition
expression_3 is incrementer or decrementer

Example:

```
#include<iostream.h>
Void main()
{
For(int i=1;i<=10;i++)
Cout <<i<<"\t";
}
```
Output:
```
1   2     3    4    5   6     7    8   9    10
```
while loop

while loop is used when we are not certain that the loop will be executed. After checking whether the initial condition is true or false. If it is to be true, only then the block is executed.

The first expression is used to initialize the index value before the while statement, the second to check whether or not the loop is to be continued using while statement and the index value incremented or decremented within the block.

General Syntax:
```
While (expression)
{
Statement 1;
Statement 2;
…………
…………
Statement n;
}
```
Example :
```
#include<iostream.h>
Void main()
{
Int i=1;
While <i<=10)
{
Cout<<i<<"\t";
I++;
}
}
```
Output:
```
1    2    3   4    5    6    7    8    9    10
```

(ii) Post test Iteration

A condition is tested only after the block is executed. This ensures that the block is executed at least once because the iteration termination condition is tested only after the execution of block.

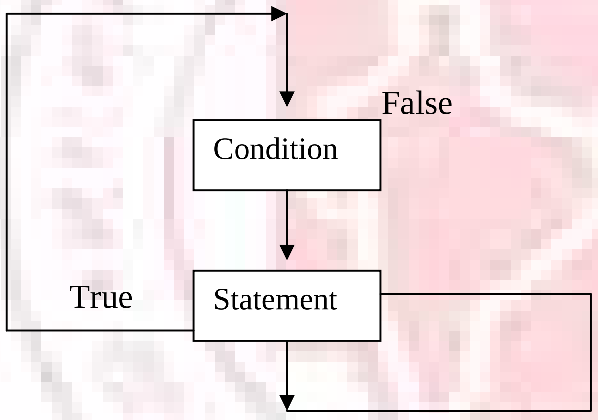
Example: do-while

Do while loop

Do while loop ensures that block is executed at least once because the iteration termination condition is tested only after the execution of the block

General Syntax:

```
Do
{
Statement 1;
Statement 2;
…………
…………
Statement n;
}
While (expression);
```



Example:

```
#include<iostream.h>
Void main ()
{
Int i=1;
Do
{
Cout <<i<<"\t";
I++;
}
While(i<=10);
}
```

Output:

```
1       2       3       4       5       6       7       8       9       10
```

3. BREAKING STATEMENTS

Loop perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.

Jumping can be done using break, continue or goto statements in any loops such as while do or for loops.

Break Statement

The break statement causes control to pass to the statement following the innermost enclosing while, do, for, or switch statement. The break statement is used to break the current iteration.

Syntax:

```
Break;
```

Example:

```
#include<iostream.h>
Void main()
{
For(int i=1;i<=10;i++)
If(i==5)
Break;
Cout<<i<<"\t";
}
```

Output:

```
1        2        3        4        5
```

Break statement is also use in switch-case structure.

Example:

```
#include<iostream.h>
Void main()
{
Int n;
Cout<<"Enter a number between 1to 4"<< endl;
Cin >>n;
Switch (n)
{
Case 1: cout<<"One";
Break;
Case 2: cout<<"Two";
Break;
Case3: cout<<"Three";
Break;
Case 4: cout<<"Four";
Break;
Default: cout<<"Invalid Input";
}
}
```

Output:

```
Enter a number between 1 to 4
2
Two
```

The output of the above program without any break statement,

```
Enter a number between 1 to 4
2
Two
Three
Four
Invalid Input
```

Continue Statement

The continue statement is used to repeat the same operations once again even if it checks the error. That is it causes the control to pass to the end of the innermost enclosing while, do, or for statement, at which point the loop continuation condition is re-evaluated.

General Syntax:

```
Continue;
```

Example 1:

```
#include<iostream.h>
Void main()
{
For(int i=1;<=10; i++)
{
If (i==5)continue;
Cout<<i<<"\t";
}
}
```

Example 2:

```
#include<iostream.h>
Void main()
{
For(int i=1;i<=10;i++)
{
If(i==5)
Continue;
Cout<,i<<"\t";
}
}
```

Output:

```
1    2     3     4     5      6     7   8    9      10
```

Goto Statement

Goto statement is used to alter the program execution sequence by transferring control to some other part of the program.

General Syntax:

Goto label;

Where label is a valid identifier used to label destination such that control could be transferred.

There are two ways of using the goto statements in a program namely conditional goto and Unconditional goto.

Conditional goto

The conditional goto is used to transfer the control of the execution from one part of the program to the other in certain conditional cases.

Example:

```
#include<iostream.h>
Void main()
{
Int x=10,y=20;
Int z;
If(x>y)
Goto en;
Z=x +y;
Cout<<z;
En:
Cout<<"Unconditional";
}
```

Output:

```
30
```

Unconditional goto

The unconditional goto statement is used just to transfer the control from one part of the program to the other part without checking any condition.

Example

```
#include<iostream.h>
Void main()
{
Int x=10,y=20;
Int z;
Goto en;
Z=x +y;
Cout<<z;
En:
Cout<<"Unconditional";
}
```

Output:

```
Unconditional
```

FUNCTIONS AND PROGRAM STRUCTURES

FUNCTIONS

A **function** is defined as a self contained block of code that performs a particular task. A complex problem may be decomposed into a small or easily manageable parts of modules called functions. Such function can called and used whenever required. Functions are very useful to read, write, debug and modify complex programs. They can also be easily incorporated in the main program. In C++, the main() itself is a function that means the main functions is invoking the other functions to perform various tasks.

Any function can call any other function. A called function can call another function and sometimes the function can call itself. A called function can be placed either before or after the calling function.

Writing function avoids rewriting the same code again and again. Separating the code into modular functions also make program easier to design and understand.

The main advantages of using a function are:

- Easy to write a correct small function
- Easy to read, write and debug a function
- Easier to maintain or modify such a function
- Small function tend to be self documenting and highly readable
- It can be called ant number of times in any place with different parameters

A function consist the thee main parts

- Function Declaration (Prototype declaration)
- Function Definition
- Function Call

FUNCTION DECLARATION

The function refers to the type of value it would return to the calling portion of the program. Any of the basic data types such as int, float char, etc. may appear in the function declaration. When a function is not supposed to return any value, it may be declared as type void, which informs the compiler not to save any temporary space for a value to be sent back to the calling program.

General Syntax:

```
<return_data_type>function_name ([data_typearg1, data_ type arg2, ., data_ type argn]);
```

Where function_ mane can be any name conforming to the syntax rule of the variables.

Arg1, arg2, … argn are Formal Arguments.

Example:

```
Void print(void);
Void pin(int, int);
Int naming (int, float);
```

FUNCTION DEFINITION

A function definition has a function header and the body of the function. A function header consist of the return data type, function name, a parentheses pair containing zero or more parameters and a body. Any parameter not declared is taken to be an int by default.

General Syntax:

Function Header

Function body

```
<return_data_type>function_name ([data_type arg, data_type arg2, . , data_ type argn])
{

Body of the function;
…………………….
……………………
Return value;
}
```

Return Statement

Return is used to terminate a function and return a value to its caller. The return statement may also be used to exit a function without returning a value. The return statement may or may not include an expression and that can appear anywhere within a function body.

General Syntax:

```
Return;
Return(expression);
```

Example:

```
Return;
Return(5);
Return(5*9);
Return(i++);
```

FUNCTION CALL

A function can be called by the main function or some other functions whenever required. A function called by itself is called Recursive function.

General Syntax:

Function_name(data_type arg1, data_typearg2, … argument);

Where arg21, arg2, …, argm are actual parameters.

ARGUMENTS

Any variable declared in the body of a function is said to be local to that function called ***local variables***. Any variable declared in the function header is said to be local to that function called ***formal arguments***. Any variable invoked in the function calling called ***actual arguments***. Other variables which are not declared either as arguments or in the function body, are considered ***global variables***.

Example:

```
#include <iostream.h>
Void print(int);                  //Prototype or Function Declaration
Int x=5;                          //Global variable x
Void main()                       //Main function
{
Int n;
Cout<<"Enter how many stars";
Cin>>n;
Print(n);                            //Function calling with the actual argument
}

Void print(int t)                    //Function with the formal argument t
{
Int                                  //local variable I of the function print
For(i=1;<=t; i++)
{
Cout<<"*"<<"\t";
}
}
```

TYPE OF FUNCTIONS

The user defined functions may classified in the following three ways based on the formal arguments passed and the return statement.

Functions are classified into three types

1. Function without argument and return statement
2. function with argument and without return statement
3. function with argument and return statement

1. Function without argument and return statement

A function is invoked without passing any format argument from the calling portion of a program and also the function does not return back any value to the called function.

Example:

```
#include <iostream.h>
Void print (void);
Void main ()
{
Print ();
}
Void print ()                    //function without arguments and return data type
{
Cout<<"\n ************************\n"; //function without return statement
}
```

Output

```
************************
```

(b) Function with argument and without return statement

A function is invoked with formal arguments from the calling portion of a program but the function does not return back any value to the calling portion.

Example:

```
#include<iostream.h>
Void print (int);
Void main ()
{
                //main function
Int n;
Cout<<"Enter how many stars";
Cin>>n;
Print (n);
                //Function calling with the actual argument
}
```

```
Void print (int t)
                    //Function with argument without return data type
{
Int I;
For (i=1 ;< =t; i++)
{
Cout<<"*"<<"\t";
}
}
```

(c) Function with argument and return statement

Function is invoked with formal arguments from the calling portion of a program which return back a value to the calling environment.

Example 1:

```
//Program for factorial of a number.
#include<iostream.h>
Long int fact (int);
Void main ()
{
Int n;
Cout<<"Enter the number"<<endl;
Cin>>n;
Cout<<"\nFactorial value is ";
Cout<<fact (n);
}
Long int fact (int t)
{
Long int f=1;
For (int i=1; i<=t; i++)
{
F=f*I;
}
Return (f);
}
```

Output:

```
Enter the number
5
Factorial value is 120
```

Example 2:

Find out the value of nCr

$$nCr = \frac{n!}{r! * (n-r)!}$$

```
//Program for factorial of a number.
```

```
#include<iostream.h>
Long int fact (int);
Void main ()
{
Int n, r;
Cout<<"Enter the n and r value"<<endl;
Cin>>n>>r;
Cout<<"\nmCr value is ";
Cout<<fact (n)/(fact(nr-)/fact(r));
}
Long int fact (int t)
{
Long int f=1;
For(int i=1; i<=t; i++)
{
F=f*I;
}
Return (f);
}
```

Output:

```
Enter the n and r value
5 2
nCr value is 10
```

ACTUAL AND FORMAL ARGUMENTS

When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the argument or the return types will be caught by the compiler at the time of compilation itself. The argument may be classified under two groups: actual arguments and formal arguments.

- **Actual arguments**
  An actual argument is a variable or an expression contained in a function call that replaces the formal parameter which is a part of the function declaration.
  Sometimes, a function may be called by a portion of a program with some parameters and these parameters are known as the actual arguments.

- **Formal arguments**

- Formal arguments are the parameters present in a function definition which may also be called as dummy arguments or the parametric variables. When the function is invoked, the formal parameters are replaced by the actual parameters.

Example:

```
#include<iostream.h>
void main ()
{
        int add (int, int);  //Prototype declaration
        int x, y, z;
        cin>>x>>y;
        z=add (x, y);      //Function calling with Actual Parameters x and y
        cout<<z;
}
int add (int a, int b)   //Function with Formal Arguments a and b
{
        return (a+b);
}
```

LOCAL AND GLOBAL VARIABLES

The values changed during the execution of a program called variable. The variable is generally classified into two types: local and global variables.

- **Local variables**

  Identifiers declared as label, const, type, variables and functions in a block are said to belong to a particular block or function and these identifiers are known as ***local variables.*** Local variables are defined inside a function block or a compound statement.

  Local variables are referred only the particular part of a block or a function. Same variable name may be given to different parts of a function or a block and each variable will be treated as a different entity.

Example:

```
void fn(int t)
{
        int I j;                        // i and j are local variables.
}
```

Global variables

Global variables are defined outside the main function block. These variables are referred by the same data type and by the same name through out the program in both the calling portion of a program and in the function block. Whenever some of the variables are treated as constants in both the main and the function block, it is advisable to use global variables.

Example:

```
int x, y=4;             // x and y are global variables
void main()
{
        test();
}
void test()
{
        cout<<y;
}
```

Output:

```
4
```

Example:

```
#include<iostream.h>
void circle (int);
const float pi=3.14;     //Global variable pi
void main()
{
circle(1);
}
void circle(int t)
{
        cout<<pi*t*t;
}
```

Output:

```
3.14
```

DEFAULT ARGUMENTS

In the function prototype declaration, the default values are given. Whenever a call is made to a function without specifying an argument. The program will automatically assign values to the parameters from the default function prototype declaration. Default arguments facilitate easy development and maintenance of programs.

General syntax:

<return_data_type> function_name (data_type arg1=value data_type arg2=value);

Example1:

```
#include <iostream.h>
Int add (int a=4, int b=6);
Void main ()
{
Int a, b;
Cin>>a>>b;
Cout<<add () <<endl;              // Passing default values
Cout<<add (a) <<endl;             // Passing one argument and the another is default
Cout<<add (a, b) <<endl;          // Passing the two arguments
}
Int add (int a, int b)
{
Return (a +b);
}
```

Output:

```
10      20
11
16
30
```

Note: The function call without argument is valid. The default arguments are given only in the function prototypes and should not be repeated in the function definition.

MULTIFUNCTION PROGRAM

A function may call one more function and so on. There is no restriction in C++ for calling the number of functions in a program. It is advisable to break a complex problem into smaller and easily manageable parts and then define a function. Control will be transferred from the calling portion of a program to the called function block. If the called function is executed successfully, then control will be transferred back to the calling portion of a program. In this type of multifunction program, transfer of control between the calling portion and a called function block is always in overhead.

General Syntax:

```
Function 1()
{
        Void function 4();
        ………………...
        ………………...
        Function 2 ();
        ………………...
         Function4 ();


}
Function2 ();
{
        Void function3 ();
        …………………
        Function3 ();
}
Function3 ()
{
        ……………….
}
Function4 ()
{
        ………………..
        }
```
Example:

```
        Void function1 (void);
        Void function3 (void);
        #include<iostream.h>
        Void main ()
        {
        Function1 ();
        Function3 ();
        }
        Void function1 ()
        {
        Void function1 ()
        {
                Void function2 (void);
                Cout<<"Function 1 called"<<endl;
                Function2 ();
        }
        Void function2 ()
        {
                Cout<<"Function 2 called"<<endl;
        }
        Void function3 ()
        {
                Cout<<"Function 3 called"<<endl;
        }
```
Output:

```
        Function 1 called
        Function 2 called
        Function 3 called
```
Note: we can not call the function function2 in main function, because of the function2 prototype is declared in function1.

SORAGE CLASS SPECIFIERS

The storage class specifier refers to how widely it is known among a set of functions in a program. In other words, how memory reference is carried out for a variable. Every identifier in C++ not only has a type such as integer, double and so on but also storage class that provides information about its visibility, lifetime and location. Normally, in C++, a variable can be declared as belonging to any one of the following groups.

1. Automatic variable

Internal or local variables are the variables which are declared inside a function. Instead, they are more often referred to as automatic due the fact that their memory space is automatically allocated as the function is entered and released as soon as it leaves. In other words, automatic variables are given only temporary memory space. They have no meaning outside the function in which they are declared. The portion of the program where a variable can be used is called the scope of that variable. Automatic variables can be declared not only at the beginning of function but also at the beginning of a compound statement (also called a block).

Local variables are given the storage class auto by default. One can use the keyword auto to make the storage class explicit but no one does, sine a declaration

General Syntax:

Storage_ class data_ type var1, var2… Varn;

Here the storage class is an automatic, so it can be written as

```
Auto int x, y, z;
Auto float a1, a2;
Auto char name1;
```

But the above declaration can also be written as

```
Int x, y, z;
Float a1, a2
Char name1;
```

However, both the declarations are same. The keyword auto used only if one desires to declare a variable explicitly as an automatic variable.

2. Register variable

Automatic variables are stored in the memory. As accessing a memory location takes time (much more time than accessing one of the machine's registers) one can make the computer to keep only a limited number of variables in their registers for fast processing whenever some variables are to be read repeatedly used, they can be assigned as register variables.

General Syntax:

register data type var1, var2 …varn;

The keyword **register** is used to declare that the storage class of the variable is register type.

For a limited number of variables it is possible to make the computer to keep them permanently in fast registers. Then the keyword register is added in their declaration.

Example:

```
Function (register int n)
{
Register char temp;
………………….
………………….
}
```

If possible, machine registers sometimes called accumulators, can be assigned to the variable n and temp, which would increase the speed. If there are not enough register variables then the request will simply be ignored.

3. Static variable

Static variables are defined within a function and have the same scope rules of the automatic variables but in the case of static variables, the contents of the variables will be retained throughout the program. It preserves the previous iteration value.

General Syntax:

Static data type var1, var2 … varn;

The static keyword is used is used to define the storage class as a static variable.

Example:

```
Static int x, y;
Static int x=100;
Static char ch;
```

4. External variable

5.

Variables which are declared outside the main are called external variables and these variables will have the same data type throughout the program, both in main and in the function.

This implicit initialization is convenient and is taken advantage of in countless programs, but explicitly initializing global variables makes programs more readable

***General Syntax*:**

Extern data type var1, var2 … varn.

RECURSIVE FUNCTION

A function which is calls itself directly or indirectly by a number of times is known as the ***recursive function***. Recursive function are very useful while constructing the data structures like linked lists, double linked lists and trees. There is a distinct difference between normal and recursive functions. A normal function will be invoked by the main function whenever the function name is used, whereas the recursive function will invoked by itself directly or indirectly as long as the given condition satisfied.

Example:

```
#include <iostream.h>
Void fun1 (void);
Void main ()
{
        …………
        …………
        Fun1 ();
}
Void fun1 ()
{
        …………
        …………
        Fun 1 ();          //Function calls recursively
}
```

PREPROCESSORS

Preprocessor is a program that modifies the C++ source program according to directive supplied in the program. The original source program is usually stored in a file. The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the complier. The preprocessor make the program easy to understand and port it from one platform to another. A preprocessor carries out the following actions on the source file before it is presented to the compiler. These actions consist of

- Replacement of defined identifiers by pieces of the text.
- Conditional selection of parts of the source file
- Inclusion of other files
- Renumbering of source files and renaming of source files itself.

The general rules for defining a preprocessor are

a) all preprocessor directives begin with #
b) the preprocessor directive is terminated not by semicolon
c) Only one preprocessor directive can occur in a line
d) The preprocessor directives may appear at any place in any source file: Outside/inside function or inside compound statements.

The common C++ preprocessor directives and their uses are

DIRECTIVE	**USES**
#include	Insert text from another file
#define	Define preprocessor macro
#undef	Remove macro definitions
#if	Conditionally include some text based on the value of the constant expression
#ifdef	Conditionally include some text based on predefined macro name
#ifndef	Conditionally include some text with the sense of the test opposite to that of #ifdef
#else	Alternatively include some text, if the previous #if, #ifdef or #ifndef tests failed
#elif	Combination of #if and #else
#endif	Terminate conditional text
#line	Give a line number for compiler messages

#error	Terminate processing early

(a) Simple macro Definitions

A macro is simply a substitution string that is placed in a program.

Example:

```
#include<iostream.h>
#define TRUE 1
#define FALSE 0
Void main ()
{
      Cout <<TRUE;
}
```

Output

```
1
```

(b) Macro with Parameters

A more complex from of a macro definition declares the names of formal parameters within parentheses, separated by commas.

General Syntax:

```
#define name (var1, var2… varn) substitution_ string
```

Example 1:

```
#define PRODUCT (x, y) {(x) * (y)}
```

Macros operate purely by textual substitution of tokens. The c++ compiler pares the source program only after the completion of the macro expansion processes are completed. Hence, care must be taken to get the desired result.

Example 2:

```
#define prod(x) x*x;
#include <iostream.h>
Void main ()
{
Cout <<prod (10);
}
Output:
100
```

(c) Other preprocessing Techniques

As the preprocessor merely substitutes a string of text for another without any checking a wide variety of substitutions are possible and hence, it is possible to make any typed source program look like another language. To illustrate the above, if someone has strong liking for Pascal and its syntax, then many Pascal symbols can be used in C++ by just including as many #define statements so as to convert them to valid C++ symbols before the compilation.

General Syntax:

#define identifiername keyword

Example:

```
#define begin {
#define End}
#define writeIn printf
#define program main
#include <iostream.h>
#include<Stdio.h>
Program ()
Begin
    WriteIn ("Pascal Output\n");
End
```

Output:

```
Pascal Output
```

(d) Conditional compilation

The preprocessor conditional compilation commands allow lines of the source text to be passed through of eliminated by the preprocessor on the basis of a computed condition.

The following are the preprocessor conditional commands

```
#if
#else
#endif
#elif
```

The above commands are used in the following way

```
#if constant expression
        Statements
#else
        Statements
#endif
```

The #elif command

The #elif command is fairly a recent addition to C++. It is supported by very few compilers only. The #elif command is like a combination of #if and #else. It is used between #if and #endif in the same way as #else but has a constant expression to evaluate in the same way as #if.

General Syntax:

```
#if constant expression
        Statements
#elif constant expression
        Statements
#elif constant expression
```

```
        Statements
#else
        Statements
        #endif
```

A header file contains the definition, global variable declaration, and initialization by all the file in a program. Header files are not compiled separately. The header file can be included in the program using the macro definition # include command. A header file can be declared as a first lin any C++ program. For example, the standard input/output statements.

The header file can be declared in one of the following ways:

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

Example: for user defined header file

```
        void test ()
        {
            int i=10;
            cout<<I;
        }
```

save the above file named test. H in the folder /te/lib

The following file use the method test

```
        #include<iostream.h>
        #include<test.h>
        void main ()
      {
        test ();
      }
```

STANDARD FUNCTIONS

Standard libraries are used to perform some predefined operations on characters, strings, etc. the standard libararies are invoked using different names such as library functions, built in functions or predefined functions. As the term library function indicates, there are a great many of them, actually they are not part of the language. Many facilities that used in C++ program need not be part of the C++ language. Most of the C++ compilers support the following standard library facilities.

- Operation on character
- Operatons on string
- Mathematical operations
- Storage allocation procedures
- Input/output operations

ARRAYS

An ***Array*** is a group of related data items that share a common name. the individual values in an array are called ***elements***. Array elements are also variables.

Arrays are sets of values of the same type, which have a single name followed by an ***index.*** Whenever an array name with an index appears in an expression, the compiler assumes that element to be of an array type.

ARRAY DECLARATION

Declaring the name and type of an array and setting the number of elements in the array is known as dimensioning the array. In the array declaration, one must define the type of the array, name of the array, number of subscripts in the array and the total number of memory locations to be allocated.

General Syntax:

[Storage_ class] data_ type array- name [expression];

Where storage_ class refers to the scope of the array variable such as external, static or an automatic. It is an optional one

Data_ type is used to allocate the type of the memory (int, float, char, etc)

Array_ name is the name of the array

Expression is used to declare the size of the memory locations required for further processing by the program which is always positive.

Example:

```
Int marks [300]; //a mark of 300 integer numbers
Static char page [8]; //a static array which consists of 8 characters
```

ARRAY INTILIZATION

The automatic array cannot be initialized, unlike automatic variables. However external and static arrays can be initialized if is desired. The initial values must appear in the same order in which they will be assigned to individual array elements, enclosed in braces and sparated by commas.

General Syntax:

Storage_ class data_ type array_ name [expression] = {element1, element2, element};

Example:

```
#include<iostream.h>
void main ()
{
        int values [7] = {10, 27, 98, 33, 44, 53, 34};
        for (int i=0; i<7; i++)
        cout<< values [i]
}
```

Output:

```
values [0] = 10
values [1] = 27
values [2] = 98
values [3] = 33
values [4] = 44
values [5] = 53
values [6] = 34
```

Note: Array index always start with 0

Program: A program to read n numbers from the keyboard to store it in an one dimensional array and to display the contents of that array elements.

```
#include<iostream.h>
void main ()
{
        int n;
        int x [100];
        cout<<”Enter how many numbers\n”;
        cin>>n;
        cout<<”\n Enter the elements”;
        for (int i=0; i<n; i++)
        cin>>x[i];
        cout<<”\n Contents of the array/n;
        for (int j=0; j<n; j++)
        cout <<x[j] <<”\t”;
}
```

Output:

```
        Enter how many numbers
        3
        Enter the elements
        56-2
        Contents of the array
        5       6           -2
```

Program: A program to read n numbers and find out the biggest number in the given array

```
#include<iostream.h>
void main ()
{
        int n; big=0;
        int x [100];
        cout<<”Enter how many numbers\n”;
        cin>>n;
        cout<<”\n Enter the elements”;
        for (int i=0; i<n; i++)
        cin>>x[i];
        cout<<”\n Contents of the array/n;
        for (int j=0; j<n; j++)
        {
        if (big<x[j[)
        big=x[j];
```

```
                }
                cout <<x[j] <<"\t";
                }
```
Output:

```
                Enter how many numbers
                3
                Enter the elements
                53-2
                The biggest number is 5
```
Program: A Program to read n numbers and to sort them in ascending order.

```
        #include<iostream.h>
        void main ()
        {
                int i, n;
                int x [100];
                cout<<"Enter how many numbers\n";
                cin>>n;
                cout<<"\n Enter the elements";
                for (int i=0; i<n; i++)
                cin>>x[i];
                cout<<"\n Contents of the array/n;
                for (int i=0; i<n; i++)
                for (int j=0; j<n; j++)
                {
                        if (x[i]>x[j];
                        {
                            int temp=x[i];
                            x[i] =x[j];
                            x[j] =temp;
                        }
                }
                cout<<"The Ascending order becomes\n";
                for (i=0; i<n; i++)
                cout <<x[j] <<"\t";
        }
```
Output:

```
                Enter how many numbers
                3
                Enter the elements
                45   3-2
                The Ascending order becomes
                -2    3    45
```

ARRAYS AND FUNCTIONS

The entire array can be passed on to a function. And array name can be used as san argument for the function declaration. Bo subscripts or square brackets are required to invoke a function using arrays.

General syntax:

```
Int sum array (int [], int max)
{
        Statements;
}
```

Program: A program to read a set of numbers from the keyboard and to sort out the given array of elements in ascending order using a function.

```
#include<iostream.h>
void main ()
{
        int i, n;
        int x [100];
        cout<<"Enter how many numbers\n";
        cin>>n;
        cout<<"\n Enter the elements";
        for (int i=0; i<n; i++)
        cin>>x[i];
        asc (x,n);
        }
        void asc (int x[], int n)
        {
        for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
        {
                if (x[i]>x[j];
                {
                    Int temp=x[i];
                    x[i] =x[j];
                    x[j] =temp;
                }
        }
        cout<<"The Ascending order becomes\n";
        for (i=0; i<n; i++)
        cout <<x[j] <<"\t";
}
```

Output:

```
Enter how many numbers
3
Enter the elements
53-2
The Ascending order becomes
-2    3    4
```

MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are defined in the same manner as one dimensional arrays, except that a separate pair of square brackets are required for each subscript. Thus, a two dimensional array will require two pairs of square brackets; a three dimensional array will require three pairs of square brackets and so on.

General syntax:

*[*storeage_clas*]*data_type array_name[expression1][expression2]…[expressionN];
Where storage_class refers to the scope of the array variable such as external, static or an automatic. It is an optional one
data_type is used to allocate the type of the memory (int,float,char,etc)
array_name is the name of the array
expression1, expression2, …, expressionN refers to the maximum size of the each array1 locations.

Example:

```
float x[10][10];
```

MULTIDIMENSIONAL ARRAYS INITIALIZATION

Similar to one dimensional array, multidimensional arrays can also be initialized, if one intend to assign some values to these elements. It should be noted that only external or static arrays can be initialized.

Example 1:

```
#include<iostream.h>
        void main()
        {
        int I, j, x[2][2]={1, 2, 3, 4};
        for(i=0; I<2; i++)
                for(j=0; j<2; j++)
                cout<<”\nValues of “<i<<”,”<<j<<”is\t”<<x[i][i];
        }
```

Output:

```
Values of 0,0 is    1
Values of 0,1 is    2
Values of 1,0 is    3
Values of 1,1 is    4
```

Example 2:

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

```
        {
        int I, j, x[0][2]={1,2};
                for(i=0; I<2; i++)
                        for(j=0; j<2; j++)
                        cout<<"\nValues of "<i<<","<<j<<"is\t"<<x[i][i];
                }
```

Output:

```
        Values of 0,0 is    1
        Values of 0,1 is    2
        Values of 1,0 is    0
        Values of 1,1 is    0
```

Program: A program to read an n x m matrix and print the elements in matrix form.

```
    #include<iostream.h>
    void main()
    {
        int I, j, ,n ,m, a[10][10];
        cout<<"\nEnter the number of rows and columns";
        of rows and columns";
        cin>>n>>m;
        cout<<"/nEnter the values one by one";
        for(i=0; i<n; i++)
        for(j=0; j<m; j++)
        cin>>a[i][j];
        cout<<"\nThe matrix become..\n";
        for(i=0; i<n; i++)
        {
                for(j=0; j<m; j++)
                        cout<<a[i][j]<<"\t";
                        cout<<endl;
        }
    }
```

Output:

```
        Enter the number of rows and columns
        2 3
        1
        2
        3
        4
        5
        6
        The matrix becomes…
        1       2       3
        4       5       6
```

CHARACTER ARRAY

The procedure for declaring character array is almost the same as for other data types such as integer or floating point. Once can declare the character array be means of alphanumeric characters.

General syntax:

*[*storeage_clas*]*data_type array_name[expression1][expression2]…[expressionN];

Where storage_class refers to the scope of the array variable such as external, static or an automatic. It is an optional one

data_type is used to allocate the type of the memory (int,float,char,etc)

array_name is the name of the array

expression is used to declare the size of the memory location required for further processing by the program which is always positive.

Example:

char name[10];

The basic structure of a character array is

J	T	H	A	S	V	I	\O

Each element of the array is placed in a definite memory space and each element can be accessed separately. The array element should end with the null character as a reference for the termination of a character array.

Initializing the character array

char color[3]= “RED”;
char name[8]= “JTHASVT”;

The character arrays always is placed in a definite memory space and each element can be accessed separately. The array element should end with the null character as a reference for the termination to accommodate the character.

Example:

```
#include<iostream.h>
void main()
{
    char name[5]= “Ravic”;
    for(int i=0; i<5; i++)
    {
    Cout<<”name[“<<I<<”]\t”<<name[i];
    }
}
```

Output:

```
Name[0]     R
Name[1]     a
Name[2]     v
```

```
        Name[3]      i
        Name[4]      c
```

Program: A program to read a line from the keyboard and to display the contents of the array on the screen

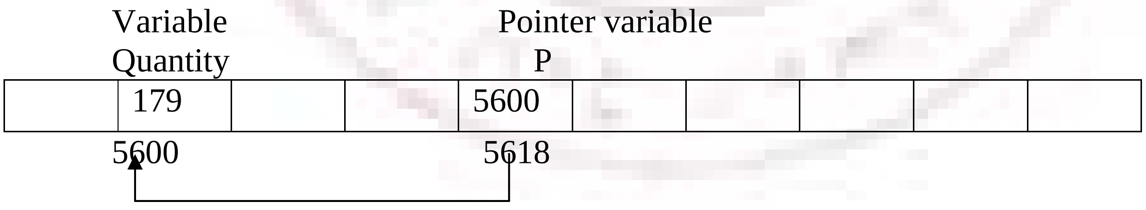
```
#include<iostream.h>
#define max 80
void main()
{
        char line[max];
        cout<<"\nEnter a line of text\n";
        cin.get(line,max,"\n); //read a line upto new line
        cout<<"Answer is"<<endl;
        cout<<line;
        int i=0;
        while(line[I]!='\o)
        ++I;
        Cout<<"\nNumber of characters\t"<<i;

}
```

Output:

```
        Enter a line of text
        Answer is
        Welcome My Dear friends
        Number of characters          23
```

POINTERS

A pointer is a variable which holds the memory address of another variable. Sometimes, only with the pointer a complex data type can be declared and accesed easily.



Advantages

- It allows to pass variables, arrays functions, strings and structures as function arguments
- A pointer allows to return structured variables from functions
- It provides functions which can modify their calling arguments
- It supports dynamic allocation and deallocation of memory segments.
- With the help of a pointer, variables can be swapped without physically moving them.

- It allows establishing links between data elements or objects for some complex data structures such as linked lists, stacks, queues, binary trees, tries and graphs.
- A pointer improves the efficiency of certain routines.

A pointer variable consists of two parts

Pointer Operator	Address Operator

Pointe Operator

Pointe variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them.

General syntax:

data-type *ptr_name;
where * tells that the variable ptr_name is a pointer variable.
ptr_name needs a memory location
ptr_name points to a variable of type data type.

Example:

Int *p;
declares the variable p as a pointer variable that points to an integer data type.

Address Operator

Once a Pointer variable has been declared, it can be made to point to a variable. An address operator can be represented by a combination of & with a pointer variable. The & is a unary operator that returns the memory address of its operand. A unary operator requires only one operand.

Example:

p=&ptr; // p receives the address of ptr

pointer Expression

a pointer is a variable data type and hence the general rule to assign its value to the pointer is same as that of any other variable data type.

Example:

```
int x, y;
int *ptr1, *prt2;
ptr1=&x;          //The memory address of variable x is assigned to the pointer //variable
                ptr1.
ptr2=*ptr1;      //contents of the pointer variable ptr1 is assigned to the variable y, not
                  The memory address.
y=*ptr1;         //The content of the pointer variable ptr1 is assigned t the
                 //variable y, not the memory address.
ptr1=&x;
```

```
    ptr2=ptr1;        //address of ptr1 is assigned to ptr2.
```

Program: A program to demonstrate pointer operations

```
include<iostream.h>
void main()
{
    int x=10, y=20;
    int *ptr1, *ptr2;
    ptr1=&x;
    ptr2=&y;
    cout<<”address of x\t”<<ptr1<<endl;
    cout<<”address of y\t”<<ptr2<<endl;
    cout<<”value of x using pointer\t”<<*ptr1<<endl;
    cout<<”value of y using pointer\t”<<*ptr2<<endl;
}
```

Output:

```
address of x            0xc40fff4
address of y            0xc40fff2
value of x using pointer     10
value of y using pointer     20
```

program: A program to assign the pointer variable to another pointer and display the contents of both the pointer variables.

```
#include<iostream.h>
void main()
{
        int x=10;
        int *prt1, *ptr2;
        ptr1=&x;
        ptr2=ptr1;
        cout<<”address of x using the pointer ptr1\t”<<ptr1<<endl;
        cout<<”address of x\t using the pointer ptr2”<<endl;
        cout<<”value of x using pointer ptr1\t”<<*ptr1<<endl;
        cout<<”value of x using pointer ptr2\t”<<*ptr2<<endl;
}
```

Output:

```
address of x using the pointer ptr1            0xc41fff4
address of x using the pointer ptr2            0xc41fff4
value of x using pointer ptr1        10
value of x using pointer ptr2        10
```

POINTER ARITHMETIC

As a pointer hold the memory address of a variable,. some arithmetic operations can be performed with pointers.

Pointer Arithmetic operators

Addition	+
Subtraction	-
Incrimination	++
Decrementation	--

Pointers are variables. They are not integer, but they can be displayed as unsigned integers. The conversion specifier for a pointer is added and subtracted.

Example:

ptr++ causes the pointer to be incremented, but not by 1.
Ptr-- causes the pointer to be decremented, but not by 1.

Program: A program to display the memory address of a variable using pointer before incrementation and after incrementation.

```
#include<iostream.h>
void  main()
{
    int x=10;
    int *ptr1;
    ptr1=&x;
    cout<<"Address of ptr1<<endl;
    ptr1++;
    cout<<"The next location after ptr1\t"<<ptr1<<endl;
```

output:

```
    address of ptr1                 0xc3bfff4
    the next location after ptr1    0xc3bfff6
```

summary of pointer arithmetic

POINTER ARITHMETIC	**DESCRIPTION**
Ptr++	Ptr=ptr+sizeof(data_type) use original value of ptr and then ptr is incremented after statement execution
++ptr	Ptr=ptr+sizeof(data_type) use original value of ptr is incremented before the statement execution
Ptr--	Ptr=ptr-sizeof (data_type) use original value of ptr and then ptr is decremented after statement execution
--ptr	Ptr=ptr-sizeof (data_type) use original value of ptr decremented before the statement execution
*ptr++	*(ptr++) retrieve the content of the location pointed to by pointer arid

	then increment ptr
*++ptr	*(++ptr) increment pointer and then retrieve the content of the new location pointed to by ptr.
(*ptr)++	Increment content of the location pointed by ptr. For pointer type content, use pointer arithmetic else use standard arithmetic
++*ptr	++(*ptr) increment the content of the location pointed to by ptr depending on the type of the content
--*ptr	--(*ptr) decrement content of the location pointed to by ptr depending on the type of the content
*ptr--	*(ptr--) retrieve the content of the location pointed to by ptr and then decrement ptr.
*--ptr	*(--ptr) decrement ptr, then retrieve the content of the new location pointed to by ptr
(*ptr)--	retrieve content *ptr of the location pointed to by ptr, then decrement the content of the location; ptr is not changed.

POINTERS AND FUNCTIONS

Pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and accessed. The use of the pointers in a function definition may be classified into two categories: Call by value and Call by value reference.

CALL BY VALUE

Whenever a portion of the program invokes a function with formal arguments, control will be transferred form the main to the calling function and the value of the actual argument is copied to the function. Within the function, the actual value copied form the calling portion of the program may be altered or changed. When the control is transferred back from the function to the calling portion of the program, the altered values are not transferred back. This type of passing formal arguments to a function is technically known as call by value.

General syntax:

```
Void main ()
{
        Void funct(int, int);
        …………………..
        …………………..
}
Void funct(int a, int b)
{
        ………………………
        ……………………....
    }
```

Program: A program to interchange two numbers using a function swap and the parameters are of that function are values.

```
#include<iostream.h>
void swap(int, int);
int x, y;            //x and y are declared as global variables.
void main()
{
    cout<<"Enter the x and y value for swapping"<<endl;
    cin>>x>>y;
    swap(x, y);
    cout<<"x value is"<<x<<endl;
    cout<<"y value is"<<y<<endl;
}
void swap(int a, int b)
{
x=b;
y=a;
}
```

Output:

```
Enter the x and y value for swapping
76
78
x value is 78
y value is 76
```

CALL BY REFERENCE

When a function is called by a portion of a program, the address of the actual arguments is copied onto the formal arguments, though they may be referred by different variable names. The content of the variables that are altered within the function block are returned to the calling portion of a program in the altered form itself, as the formal and the actual arguments are referencing the same memory location or address called Call by Reference.

When an argument is passed by value, the data item is copied to the function. Thus, any alteration made to the data item within the function is not carried over into the calling routine. When an arguments passed by reference, the address of the data item is passed to the function. The contents of that address can be accessed freely, either within a function or within a calling routine. Moreover, and change that is made to the data item will be recognized in both the function and the calling portion of the program. Thus, the use of the pointer as a function arguments permits the corresponding data item to be altered globally form with the function.

General syntax:

```
Void main ()
```

```
{
        Void funct(int *x, int *y);
        Int x=10; y=20;
        …………………..
        …………………..
        Funct(&x, &y)       //call by Reference
}
Void funct(int *x, int *y)
{
        ………………………
        ……………………....
}
```

Program: A program to interchange two numbers using a function swap and the parameters are of that function are values.

```
#include<iostream.h>
void swap(int*, int*);
void main()
{
    int x, y;
    cout<<"Enter the x and y value for swapping"<<endl;
    cin>>x>>y;
    swap(&x, &y);
    cout<<"x value is"<<x<<endl;
    cout<<"y value is"<<y<<endl;
}
void swap(int a, int b)
{
Int temp;
temp=*a;
*a=*b;
*b=temp;
}
```

Output:

```
Enter the x and y value for swapping
98
23
x value is 23
y value is 98
```

POINTERS TO FUNCTIONS

A pointer to a function must be declared to be a pointer to the data type returned by the functions, like void, int, float and so on. In addition, the argument type of the function must also be specified when the pointer is declared. The argument declaration is a list of formal arguments, separated by commas and enclosed in parentheses.

General Syntax:

return_ type (*variables)(list of parameters);

Example:

Void (*ptr)(float float, int); //a pointer to a function return void and takes the formal arguments of two float and one int.

Float (*ptr)(char, double, int, flat); //returns a floating point value and taks the formal arguments of char, double, int and float.

Program:

```
#include <iostream.h>
Void main ()
{
  Float add (float, float, ); //Function declaration
  Float (*ptradd)(float, float); // pointer to function declaration
  Ptradd &add ;
  Int a, b;
  Cin>>a>>b;
  Cout <<(*ptradd)(a, b);
}
Float add (float x, float y)
{
    Return (x +y);
}
```

Output:

```
  4 7
  11
```

PASSING A FUNCTION TO ANOTHER FUNCTION

C++ allows a pointer to pass one function to another as an argument.

General Syntax:

Return _ type function_ name (ptr_ to_ fn (other arguments);

Example:

Float calculation (float(*) (float, float), float, float);

When the function calculation return a type float and takes the formal argument of a pointer to another function and two other float types. As a pointer to function declaration itself is a pointer data, it returns a type float and takes a formal argument of two floating point values.

Program:

```
#include <iostream.h>
Void main ()
{
    Float add (float, float, ); //Function declaration
    Float action(float (*)(float, float,), float, float);
    Float (*ptradd)(float, float); // pointer to function declaration
```

```
        Ptradd &add ;
        Int a, b;
        Cin>>a>>b;
        Cout <<(*ptradd)(a, b);
    }
    Float add (float x, float y)
    {
        Return (x +y);
    }
    Float action (float(*ptradd)(float, float), float x, float y)
    {
        Float answer;
        Answer=(*ptradd)x, y);
        Return (answer);
    }
```

Output:

```
        4 5
        9
```

POINTERS AND ARRAYS

Arrays are similar to pointers except pointer is a variable that can appear on the left side of an assignment operator. The array name is a constant and cannot appear on the left side of an assignment operator. In all other respects, both the pointer and the array version are the same.

Pointer and One Dimensional array

In C++, pointers and one dimensional arrays have a close relationship. Consider the following valid declaration,

```
        Int value[10];
        Int *ptr;
```

Where the array variable value is an array type, and the address of the first element can be declared as value[0]- which holds the address to the zeroth element of the array value. The pointer variable ptr is also an address, so the declaration value[0] and ptr is same as both hold address.

The following is valid assignment

```
        Ptr=&value[0];
```

The address the zeroth element is assigned to a pointer variable ptr.

If the pointer is incremented to the next data element, then the address of the incremented value of the pointer will be same as the value of the next element.

```
        Ptr+1 equals value[1]
        Ptr+ 2 equals value[2]
        ……………………..
        ……………………..
        Ptr+n equals value[n]
        And also
        *ptr equals & value[0]
```

Program: A program to display the content of an array using a pointer arithmetic.

```
#include<iostream.h>
Void main ()
{
        Int a [4]={12, 23, 34, 45};
        For(int i=0;i<<4; i++)
        Cout<<"value="<<*((a) +(I))<<endl;
}
```

Output:

```
        Value =12
        Value =23
        Value =34
        Value =45
```

Pointer and Multidimensional Array

A pointer to an array contains the address of the first element. In an one dimensional array, the first element is &[0]. In a two dimensional array, it is

&value [0][0]

Example:

```
                Int value [][];
                Int *ptr;
                Ptr=value;
```

The address of the zeroth row and zeroth column of the two dimensional array value is assigned to the pointer variable value.

Suppose, if ptr++ is written, the pointer variable will be incremented to the next data element in the two dimensional array that is equal to value[0][1] because a two dimensional array is stored by rows. So the following equality is true.

Ptr+1 =&value[0][1];

Example:

```
                float value[20[30]
                float *ptr 1;
                ptr=&value[0][0];//Pointer initialization
                ptr+4 = &value [0][4];
                [tr+30 = &value[1][0];
```

If s is a 2 by 3 array as defined above, then the expression s[1[2] is expressed as *(*(s+1)+2) which is evaluated in the following order

S

S+1

*s(s+1) +2

((s+1)+2)

Program:

```
#include<iostream.h>
Void main ()
{
    Int a[2][3]={{11, 12, 13}, {14, 15, 16}};
    Int *ptr;
    Int I, j, n, m, temp;
    N=2;
    M=3;
    Cout<"\n Contents of the array"<<endl;
    For(i=0;i<n; i++)
            For (j=0;j<m; j++)
            Cout<<*(*a+ i)+j)<<"\t";
}
```

Output:

```
        Contents of the array
        11  12   13    14   15   16
```

POINTERS AND STRINGS

Many string operations in C++ are usually performed by using pointers to the array and then using pointer arithmetic. As strings tend to be accessed strictly in sequential order, pointers use the obvious choice. Strings are one dimensional arrays of type char. A string is terminated by null character or '\0' String constants are written in double quotes.

General Syntax:

```
            Data_ type *pointer_ variable="value";
```

Example:

Char *s="Hello world" //s is a character pointer pointing to the characters Hello world

Program: A program to read a string and print the characters one by one.

```
#include<iostream.h>
Void main ()
{
  Char *ptr;
  Cin>>ptr;
  While(*ptr! =NULL)
  {
          Cout<<*ptr<<"\t";
          Ptr++;
  }
}
```

Output:

```
            Welcomes
            W      e      l      c      o      m      e
```

ARRAYS OF POINTERS

The pointers may be arrayed like any other data type.

General Syntax:

```
        Data_ type *pointer_ name[size];
```

Example:

The declaration for an integer pointer array of size 10 is

```
Int *ptr[10];
```

Makes

Ptr[0], ptr[1], …ptr[10] an array of pointers.

Example2:

```
Int [10],b[10],c[20];
Int *ptr[4];
```

Where ptr is an array of pointers that can be used to point to the first elements of the arrays a, b and c,

The following declaration is valid for the two dimensional arrays,

```
Char *text [row][col];
```

For array of characters to pointes.

Example:

```
Char *name[10][15];
Name[1][]= “this”;
Name[2][]= “World”;
…………………..
…………………..
```

Program: A program to display the contents of pointers using an array of pointers.

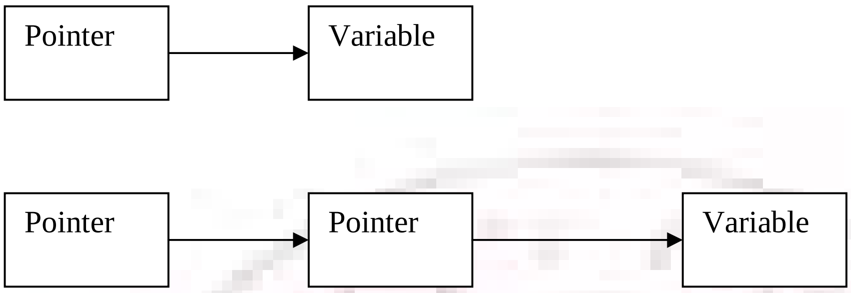
```
    #include<iostream.h>
    Void main ()
{
    Char *ptr[3];
    Ptr[0]=”ravic”;
    Ptr[1]= “raju”;
    Ptr[2]= “arul”;
    Cout<<”contents of pointer 1\t”<<ptr[0]<<endl;
    Cout<<”contents of pointer 1\t”<<ptr[1]<<endl;
    Cout<<”contents of pointer 1\t”<<ptr[2]<<endl;
}
```

Output:

```
    contents of pointer 1 ravic
    contents of pointer 2 raju
    contents of pointer 1 arul
```

POINTERS TO POINTERS

An array of pointers is the same as pointers to pointers. As n array of pointers is easy to index because the indices themselves convey the meaning of a class of pointers. However, pointers to pointers van be confusing. The pointer to a pointer is a form of multiple of indirections or a class of pointers. In the case of a pointer to a pointer, the first pointer contain the address of the second pointer, which points to the variable that contain the values desired. Multiple indirection can be carried on to whatever extent desired, but there are a few cases where more pointer o a pointer is needed



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional * in front of the variable name. The declaration informs the compiler that the next
int **ptr2;
Where ptr2 is a pointer which holds the address of the another pointer.

Program: A program to declare the pointer to pointer variable and to display the contents of these pointers.

```
#include<iostream.h>
Void main ()
{
    Int value= 100;
    Int *ptr1, **ptr2;
    Ptr1=&value;
    Ptr2=&ptr1;
    Cout<<"Pointer 1 value"<<*ptr1<<endl;
    Cout<<"Pointer 2 value"<<*ptr2<<endl;
}
```

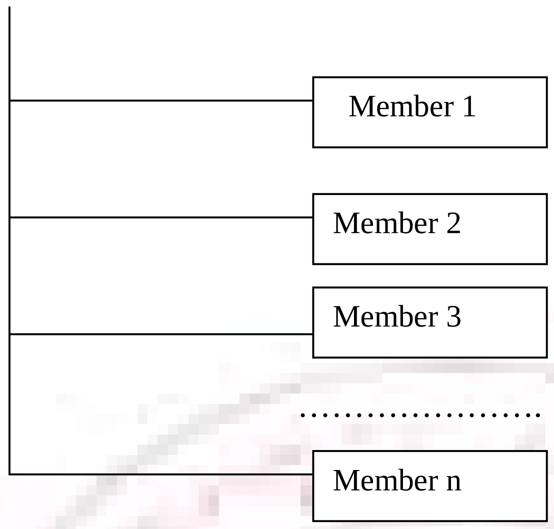
Output:

```
Pointer 1 value 100
Pointer 2 value 100
```

STRUCTURES

A collection of heterogeneous data types can be grouped to form a structure. The entire collection can be referred to by a structure name. the individual component which are called fields or members can be accessed and processed separately. There are two important distinctions between arrays and structures. One is, all the elements of an array have the same type, whereas in a structure the components or fields can have different data types.

Another distinction is that a component of an array is referred to by its position, whereas each component of a structure has a unique name. simply say that it is an user defined data type.



General Syntax:

```
            [storage_ class] struct user_ defined_ name
            {
                  Data_ type member1;
                  Data_ type member2;
                  …………………….
                  …………………….
                  Data_type memberN;
            }[s1, s2, .. , sN];
```

Where storage _ class refers to the scope of the array variable such as external, static or an automatic. It is an optional one

Struct is a keyword
Data_ type is used to allocate the type of the memory (int, float, char, etc)
Member1, member2, .., memberN are member of the structure.
S1, s2, …, sN are structure variables.

Assigning values to members of the structure

```
Structure_ variable.member1=value;
Structure_ variable.member2=value;
……………………………………
……………………………………
Structure_ variable.member3=value;
```

Program1: A program to assign some values to the member of a structure and to display the structure members' values.

```
Struct data{
      Int day;
      Int month;
      Int year;
}s1;
Si.day=25;
S1.month=9;
S1.year-76;
Void main ()
{
        Cout<<s1.day<<endl;
        Cout<<s1.month<<endl;
        Cout<<s1.year<<endl;
}
```

Output:

```
25
9
76
```

INTIALIZATION OF STRUCTURE

A structure must be initialized whenever it must be either static or external.

General Syntax:

```
                [storage_ class]struct user_ defined_ name
                {
                          Data_type member1;
                          Data-type member2;
                          …………………….
                          …………………….
                          Data_type memberN;
                }s1={member1 Value, member2value,…,memberN Value};
```

Example:

```
                Struct student
                {
                        Int rollno;
                        Float avg;
                }s1={21, 98.2};
```

Program: A program to initialize the members of a structure and to display the contents of the structure on the screen.

```
    #include<iostream.h>
    Struct student
    {
        Int rollno;
        Float mark;
    }s1={100,54};
    Void main ()
    {
          Cout<<"Roll Number\t"<<s1.rollno<<endl;
          Cout<<"Mark\t"<<s1.mark<<endl;
```

}

Output:

```
                    Roll Number 100
                    Mark 54
```

Note: A field or member of a structure is a unique name for the particular structure. The same field or member name may be given to other structures with different data types.

Example:

```
            Struct first
        {
            Int a;
            Float b;
        };
        Struct second
        {
            Int a;
            Float b;
        }
```

In the above two structures namely first and second have three members and all the two are of different data types but have same names

.

FUNCTIONS AND STRUCTES

A function is a very powerful technique to decompose a complex problem into separate manageable parts or modules. Each part is called a function and is very much used to convert a complicated modules. Each part is called a function and is very much used to convert a complicated program into a very simple one. As functions can be compiled separately, they can be tested individually and finally invoked in to a main program as a whole.

A structure can be passed to a function as a single variable. the scope of a structure declaration should be an external storage class whenever a function in the main program is using a structure data types. The field or member data should be same throughout the program either in the main or in a function.

Example:

```
            Struct student
            {
                    Int rollno;
                    Float mark;
            };
            Void main ()
            {
                    Void display (struct student s);
                    Struct student s1;
                    ………
                    ………
```

```
        }
        Void display  (struct student s)
        {
                ………
                ………
        }
```

Program: A program to display the contents of a structure using function definition.

```
        #include <iostream.h>
        Struct student
        {
                Int rollno;
                Float mark;
        };
        Void main ()
        {
                Void display (struct student s);
                Struct student s1={1000, 54 44}
                Display (s1);

        }
        Void display  (struct student s)
        {
                Cout<<"Roll Number \t"<<s.rollno<<endl;
                Cout<<"Mark \t "<<s.mark<<endl;
        }
```

Output:

```
                        Roll Number 1000
                        Mark    54. 44
```

ARRAYS OF STRUCTURES

An array is a group of identical data which are stored in consecutive memory locations in a common heading or a variable. a similar type of structure placed in a common heading or a common variable name is called arrays of structures.

Example:

```
        Struct  student
        {
                Int rollno;
                Float avg;
        }s1 [3];
```

A three user defined type contiguous memory location are allotted for the structure variable. s1. each record may be accessed and processed separately like individual elements of an array.
This is

```
        S1 [0].rollno=value;
        S1[0].avg=value;
        ………………..
        ………………..
```

Initialization of arrays of structures

A structure can be initialized in the same way as that of array data when a structure must be declared as either static or external.

```
Struct student
{
        Int rollno;
        Float avg;
}s1[3]={
              {10001, 89.2},
              {10002, 93.1}
              {10003, 8.2 }
        };
```

Program: A program to read and display some members of an array of structures and to display the contents of all the structures.

```
#include<iostream.h>
Struct student
{
      Int rollno;
      Float mark;
}
Void main ()
{
      Int i;
      Struct student s[3];
      For(i=0;i<3;i++)
      {
                Cout<<"\n Enter the roll no";
                Cin>>s[i].rollno;
                Cout<<"\nEnter the mark";
                Cin>>s[i].mark;
      }
      Cout<<"\nThe Student details …\n";
      For(i=0;i<3;i++)
      {
                Cout<<s[i].rollno<<"\t"<<s[i].mark<<endl;
      }
}
```

Output:

```
      Enter the roll no 1001
      Enter the mark 45
```

```
Enter the roll no 1002
Enter the mark 23
Enter the roll no 1003
Enter the mark 56
The Student details …
        1001     45
        1002     23
        1003        56
```

ARRAYS WITHIN A STRUCTURE

So far, the discussion has been limited to members of a structure which have been declared as an ordinary data type such as char, int, float only. However, a member of a structure can be an array data type also.

General Syntax:

```
[Storage_ class] struct user _defined_ name
{
Data_type member1;
Data_ type member2 [size]; //data member contains an array of elements
          ……………………..
          ……………………..
          Data_ type memberN;
}
```

Example:

```
Struct student
{
        Int rollno;
        Float mark [3]; //It is used to store 3 marks of a student
        };
```

Program: A program to declare the array data type within a structure and to initialize some of the structures and to display the content of the structures.

```
     #include <iostream.h>
Struct student
{
        Int rollno;
        Char name [20];
        Float marks [3];
};
Void main ()
{
        Int i;
        Struct student s;
                Cout<<"\n Enter the roll no";
                Cin>>s.rollno;
                Cout<<"\nEnter the name";
                Cin>>s. name;
                Cout<<"\nEnter the marks";
                For (i=0;i<3;i++)
                {
                Cin>>s. marks[i];
                }

            Cout<<\n The Student details …\n";
                Cout<<s.rollno<<"\t<<s. name<<endl;
                For (i=0;i<3;i++)
                {
                Cout<<"Mark" <<i+1<<":"<<s. marks [i]<<endl;
                }

}
```

Output

```
                Enter the roll no 10001
                Enter the name jthasvi
                Enter the marks 67 78 89
                The Student details …
                10001 jthasvi
                Mark 1:67
                Mark 2:78
                Mark 3:89
```

STRUCTURES WITHIN A STRUCTURE (NESTED STRUCTRE)

A structure use a member of another structure. When a structure is declared as the member of another structure, it is called as a nested structure or structure within a structure. The individual elements in a nested structure first represent the structure variable name and the first structure and then the filed name of the first structure.

To assign values

Structure_ var_ name. structure _ var_ name. fieldname= value

General Syntax:

```
                Struct one
                {
                        …………….
                        …………….
                }
                Struct second
                {
                        Struct one o;
                        …………..
                        …………..
                };
```

Program: A program to read and display a set of rollno, date of birth of a student where the date of birth consists of three members such as day, month and year, as a separate structure.

```
    #include<iostream.h>

    Struct date
    {
      Int day;
      Int mouth;
      Int year;
    };
    Struct student
    {
      Int rollno;
      Struct date d;
    };
Void main ()
{
      Int i;
      Struct student s;
                Cout<<"\n Enter the roll no";
                Cin>>s. rollno;
                Cout<<"\nEnter the Date of Birth";
                Cin>>s. day>>s. d. month>> s. d. year;
                Cout<<"\n Details …\n";
                Cout<<s. rollno<<endl;
                Cout<<"Date of Birth: "<<s. d. day<<"-"<<s. d. month<<"-"<<s. d. year;
}
```

Output:

```
    Enter the Roll no 10001
    Enter the Date of Birth 10 10 1976
    Details …
    10001
    Date of Birth: 10- 10- 1976
```

POINTERS AND STRUCTURES

A pointer is a variable which hold the memory address of a variable of basic data type such as int, float or sometimes an array. A pointer can be used to hold the address of a structure variable too. The pointer variable is very much used to construct complex data bases using the data structures such as linked lists, double linked lists and binary trees.

General Syntax:

*[*Storage_ class*]* struct user _defined_ name

```
{
Data_type member1;
Data_ type member2;
          ……………………..
          ……………………..
          Data_ type memberN;
};*ptr1;
```

Where ptr1 is a pointer variable holding the address of the structure and is its having some members.

The pointer structure variable can be accessed and processed in one of the following ways:

(*struct_ name).field_ name=variable;

The parenthesis are essential because the structure member period (.) has a higher precedence over the indirection operator (*). The pointer to structure can also be expressed using dash (-) followed by the greater than sign (>).

General syntax:

```
Structure_ name-> field_ name=variable;
```

Example1:

```
Struct student
{
        Int rollno;
        Int mark;
}*st;

*(st).rollno=10001;
*(st).mark=87;
```

Example 2:

```
#include<iostream.h>
Void main ()
{
Struct student
{
        Int rollno;
        Int mark;
};
Struct student *st;
St->rollno=10001;
St->mark=40
Cout<<st->rollno;
Cout<<st->mark;
}
```

Output:

```
10001 40
```

UNIONS

We know that structure is a heterogeneous data type which allows to pack together different types of data values as a single unit. Union is also similar to a structure data type with a difference in the way the data is stored and retrieved. The union stores values of different types in a single location.

A union will contain one of the many different types of values (as long as only one is stored a time). Union holds only one value for data type. If a new assignment is made, the previous value is automatically erased.

General Syntax:

```
[Storage_ class] struct user _defined_ name
{
        Data_type member1;
        Data_ type member2;
        ……………..
        ……………..
        Data_ type memberN;
} [u1, u2 …, un];
```

Where storage_ class refers to the scope of the array variable such as external, static or an automatic. It is an optional one

Data_ type is used to allocate the type of the memory 9int, float, char, etc)
Union is used to declare the union data type.
Member1, member2… memberN are members of the union.
U1, u2, …, un are Union variables.

Note: A union may be a member of a structure and a structure may be a member of a union moreover, structures and unions can be mixed freely with arrays.

Example:

```
Union student
{
        Int rollno;
        Float mark;
};
```

PROCESSING WITH UNION

A period operator (.) is used in between union variable name and the field name. once a union type is defined, variables for the union data type can be declared.

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

```
{
        Int ch;
        Int ch1;
  } u1;
  Void main ()
  {
        U1. ch=12;
        U1. ch1=18;
        Cout<<"\n Roll Number"<<u1. ch <<endl;
        Cout<<"\n Mark"<<u1. ch1 <<endl;
  }
```

Output:

```
        Roll Number 18
        Mark 18
```

Initialization of Unions

Static and external structures can be initialized when they are defined, and it may seem reasonable to allow the same for unions. However, a union has only cone active member at any given time and it is up to the programmer to keep track of the active member, as this information is not inherently stored with the union itself. Although pointers to unions may be used just like pointers to structures. Unions themselves may not be passed as a function arguments used in assignment statements or returned by a function. A variable may be a pointer to a union first as a pointer can point to a structure.

```
Union student
{
    Int ch;
    Int ch1;
}*u1;
```

The members can be referred by using the pointer operator

```
    U1->ch;
    U1->ch1;
```

A union can be a member of a structure and it can appear as any member of the structure. Whenever, a union is declared as a member of a structure, it should not be first member, but the last one.

Program: A program to declare a member of an union as a structure data type and to display
the contents of the union.

```
#include <iostream.h>
Struct date1 {
    Int dd;
    Int mm;
    Int yy;
};
```

```
Struct date2
{
    Int dd;
    Char mm[5];
    Int yy;
};
Union date
{
    Union date1 d1;
    Union date2 d2;
}d;
Void main ()
{
    d.d1.dd=11;
    d.d1.mm=10;
    d.d1.yy=98;
    cout<<"\nDate:"<<d.d1.dd<<"-"<<d.d1.mm<<"-"<<d.d1,yy<<endl;
    }
```

Output:

```
    Date: 11-10-98
```

BIT FIELDS

A bit field is a special type of structure member, it holds several bit fields, can packed into an int. While bit fields are variables, they are defined in terms of bits rather than character or integers. Bit fields are useful for maintaining single or multiple bit flags in an int without having to use logical AND and logical OR operations to set and clear them. They can also assist in combining and dissecting bytes and words that are sent to and received from external devices.

The formal declaration of a bit field is same as the declaration of a structure, but there is a difference in accessing and using a bit field in a structure. The number of bits required by a variable must be specified and followed by a colon while declaring a bit field. The bit fields can be signed or unsigned integers, from 1 to 16 bits in length. The number of bits will depend on the machine being used. The bit field is very useful with data items where only a few bits are required to indicate a true or condition. Secondly, the bit field is used to save the memory space. The number of bits required by each variable is declared in a structure.

General Syntax:

```
[Storage_ class] struct user _defined_ name
{
Data_type member1 field_ size;
Data_ type member2 field_ size;
………………
………………
        Data_ type memberN: field_ size;
```

Where field_ size is number of bits for the data members.

Example:

```
Struct data
{
        Int dd:1; //bit field operator
        Int mm:2;
        Int yy:3;
}d;
```

5	4	3	2	1	0	**Bits**
dd	mm		yy			

The entire structure bits is a single 16 bit word, dd takes up 1 bit and mm takes up 2 bits and yy takes up 3 bits. The way of accessing a bit field in a structure is similar to accessing another structure field. The period operator (.) is used to access a bit field of a structure.

Program: A program to demonstrate the bit field operator

```
#include<iostream.h>
Struct date
{
    Int dd: 1; //bit field operator
    Int mm;
    Int yy;
}d;
Void main ()
{
    d. dd=10;
    d. mm=10;
    d. yy= 98;
    cout<<”\nDate: “<<d. dd <,”-“<<d. mm<<”-“<<d. yy<<endl;
}
```

Output:

```
Date: 0-10-98
```

Note: we can not use scanf or cin to read values into a bit field. Instead, one has to read into a temporary variable and then assign its value to the bit field. Even the bit fields may be accessed in structure using a pointer operator indirection operator.

TYPEDEF

The typedef is used to define new data items that are equivalent to the existing data types. Once a user defined data is declared, then new variables, arrays, structures, and so on can be declared in terms of this new data types.

General Syntax:

Typedef data_ type new Type

Where typedef is a keyword for declaring the new items and data type is an existing data type being converted to the new name (new/Type).

Example:

```
        Typedef int integer;
        Typedef struct student stu;
        Typedef struct student
        {
                Int rollno;
} stud;
Stud;
```

Program: A program to define the variable using typedef and to display the contents of the variable .

```
#include<iostream.h>
Void main ()
{
      Typedef int integer;
      Integer I =1000;
      Cout<<I;
}
```

Output:

```
            1000
```

ENUMERATION

An enumeration data type is a set of values represented by identifiers called enumeration constants. The enumeration constants are specified when the type is defined.

General Syntax:

```
        Enum user_ defined _name
        {
                Member 1;
                Member2;
                ………...
                ………...
                memberN;
        }
```

Where enum is a keyword for defining the enumeration data type and the braces are essential. The members of the enumeration data type such as member1, member2, …, memberN are individual elements.

Example:

```
        Enum sample {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

Program: A program to declare the enumeration data type and to display the integer

```
    #include<iostream.h>
    Enum day { Sun, Mon, Tue, Wed, Thu, Fri, Sat};
    Void main ()
```

```
{
        Cout<<"Sunday ="<<Sun<<endl;
        Cout<<"Monday ="<<Mon<<endl;
        Cout<<"Tuesday ="<<Tue<<endl;
        Cout<<"Wednesday ="<<Wed<<endl;
        Cout<<"Thursday ="<<Thu<<endl;
        Cout<<"Friday ="<<Fri<<endl;
        Cout<<"Saturday ="<<Sat<<endl;
}
```

Output:

```
        Sunday= 0
        Monday= 1
        Tuesday= 2
        Wednesday=3
        Thursday= 4
        Friday= 5
        Saturday = 6
```

Example1:

Enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat} d= Jan;
Returns an error message because Jan is not enumeration constant.

Example 2:

Enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
The enumeration constant starts with 100.

Example 3:

Enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

The enumeration constant values becomes

Sun is 0
Mon is 1
Tue is 100
Wed is 101
Thu is 102
Fri is 103
Sat is 104

CLASS AND OBJECTS

A class is a user defined data type which holds both the data and functions. The internal data of a class is called **member data** (or **data member**) and the functions are called **member functions**. The member functions mostly manipulate the internal data of a class. The member data of a class should not normally be addressed outside a member function. The variables of a class are called **objects** or **instance of a class.**

In simply say that a class is a collection of objects of similar type.

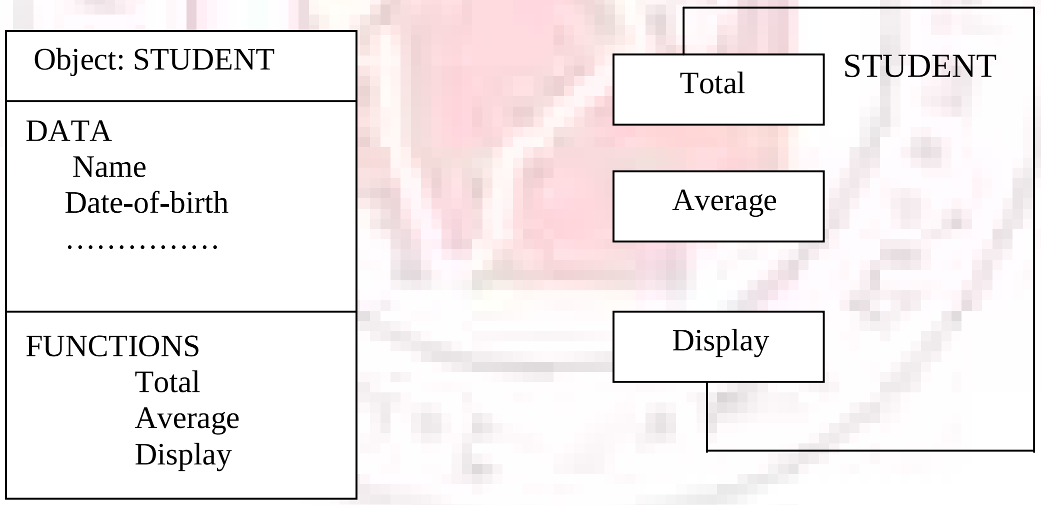
Example:

```
//mange, apple and orange are members of the class fruit.

Class fruit
{
        ………..
        ………..
};
Fruit mango, apple, orange;
```

The class construct provides support for data hiding, Abstraction, Encapsulation, Single Inheritance, Multiple inheritance, Polymorphism and public interface functions (methods)For passing message between objects.

(a) OBJECT
(b)

Objects are the basic runtime 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 must handle. The may also represent user-defined data such as vectors, time and lists.



(b) DATA ABSTRACTION, HIDING AND ENCAPSULATION

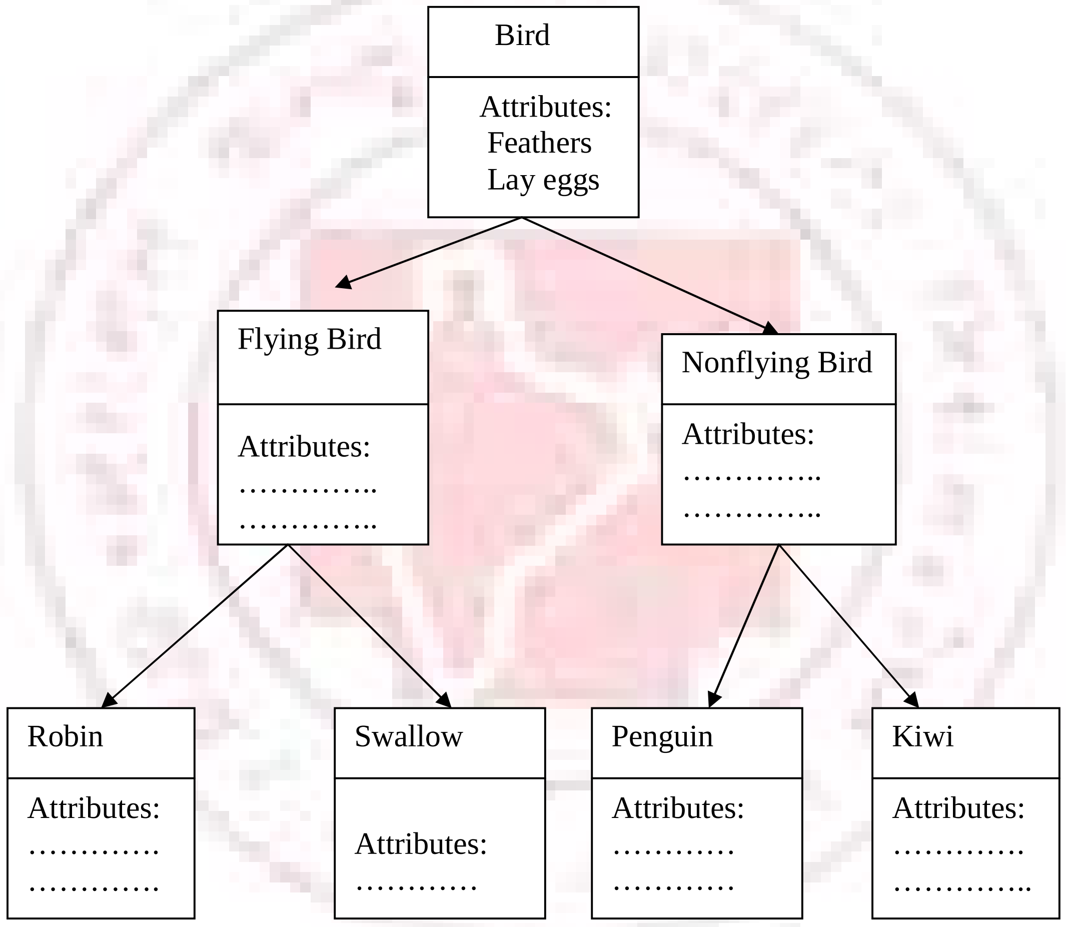
The wrapping up of data and functions into a single unit (called class) as known as **Encapsulation**. The data is not accessible to the outside word and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from access the program is called Data hiding.

Classes use the concept of abstractions and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes. They ecapsulate all the essential properties of the objects that are to be created. Since the classes use the concept Data abstraction, they are known as Abstract Data Types (ADT).

(C) INHERITANCE

Inheritance is the process by one class can acquire the properties of objects of another class. It is supports the concept of hierarchical classification.

Example:
The bird robin is a part of the class flying bird which is again a part of the class bird.



(d) POLYMORPHISM

Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances. The behavior depends upon the types of data used in the operation.

Polymorphism plays an important role on allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may diff. polymorphism is extensively used in implementing inheritance.

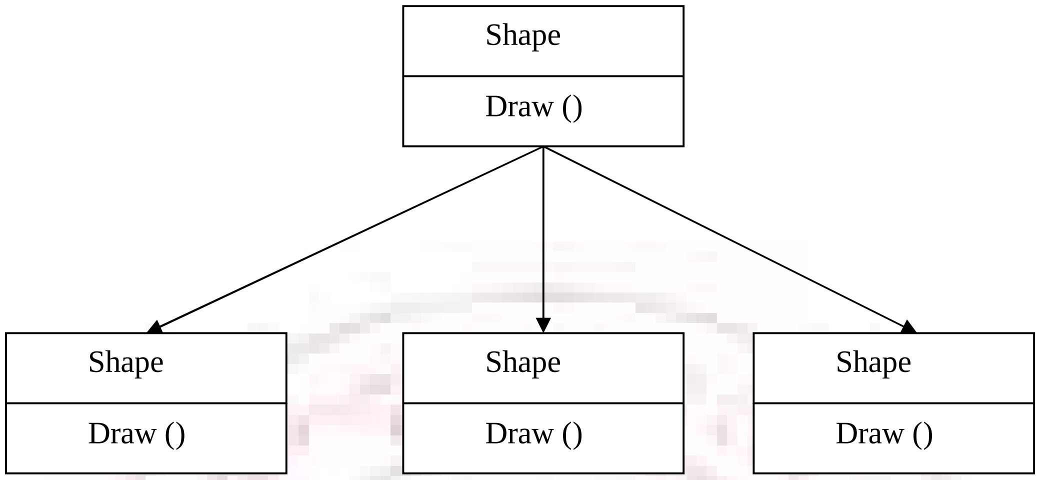


Fig. Polymorphism

(e) DYNAMIC 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. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

(f) MESSAGE COMMUNICATION

An object-oriented program consists of a set of objects that communicate with each other .the process of programming in an object-oriented language therefore involves the following basic steps

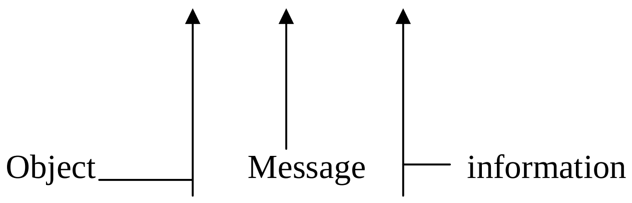
1. Creating classes that define objects and their behaviour.
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people as messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real word counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.

Example:

Employee salary (name);



BENEFITS OF OOP

OPP offers several benefits to both the program designer and the user. Object. Orientation contributes to the solution of many problems associated with the development quality of software products. The new technology promises greater programmer productivity better quality of software and lesser maintenance cost.

The principal advantages are

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes
2. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
3. the principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
4. It is possible to have multiple instances of an object to co-exist without any interference.
5. It is possible to map objects in the problem domain to those objects in the program.
6. It is easy to partition the work in a project on objects.
7. the data centered design approach enables us to capture more details of a model in implementable from.
8. Object-oriented systems can be easily upgraded from small to large systems.
9. Message passing techniques' for communication between objects makes the interface descriptions with external systems much simpler.
10. Software complexity can easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. Developing a software that is easy to use and makes it hard to build.

DECLARATION OF A CLASS

A class is a user defined data type which consists of two sections, a private and a protected section that holds data and a public section that holds the interface operations.

A class definition is a process of naming a class and data variables, and methods or interface operations of the class. In other words, the definition of a class consists

a) Definition of a class.

b) The internal representation of data structures and storage.

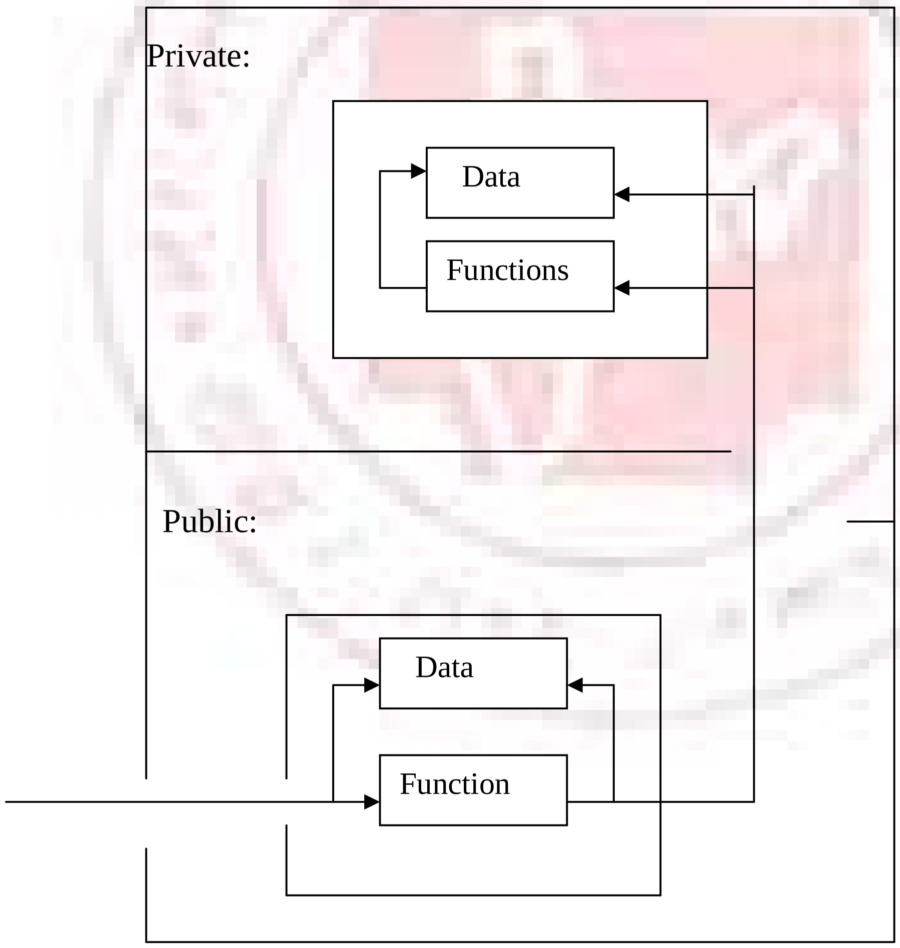
c) The internal implementation of the interface.

d) The external operations for accessing and manipulating the instance of the class.

A class declaration specifies the representation of objects of the class and the set of operations that can be applied to such objects.

Class members can be one of the following member lists:

- Data
- Functions
- Classes
- Enumerations
- Bit fields
- Friends
- Data type names



Example 1:

Class item

{

```
                Int itemcode;
                Float rate;
        };
```
Example:

```
    Class student
    {
        Int rollno;         //Data member declaration private by default.
        Int mark;
        Public:
            Void getdata();//Prototype declaration.
            Void putdata();
    };
```

MEMBER FUNCTIONS

A function declared as member of a class is called as a member function. Member functions are mostly given the attributes of public because they have to be called outside the class either in a program or in a function. The member functions of a class are designed to operate upon three data types. It can typically be classified into three types.

1. Manager function
2. Accessor functions
3. Implementor functions

1. Manager function

Manager functions are used to perform initialization and clean up of the instance of the class objects. Some of the examples for the manager functions are constructor and destructor functions.

2. Accessor function

The accessory member functions are the constructor functions that return information about an object's current state. An example for the accessor functions is a const member functions.

3. Implementor functions

These are the functions that make modifications to the data members. These functions are also called as mutators.

A class contains not only a data member but also a function which are called methods, it must be defined before it is used.

Example:

```
    Class sample
    {
        Private:
```

```
        Int x;
        Int y;


    Public:
        Void readData ()
        {
        Cin>>x>>y;
        }
        Void display ()
        {
        Cout<<x<<y;
        }
    };
```

The member functions getData () and display () are defined quite normally within the class declaration. The member functions can more complex as they can have local variable, parameters, etc.

Defining a member function of a class outside its scope

It is permitted to declare the member functions either inside the class declaration or outside the class declaration. A member function of a class is defined using the Scope operator (::).

General Syntax:

Return_ type class_ name :: member_ functions(arg1, arg2, …, argN)

Note that the type of member function arguments must exactly match with the types declared in the class definition of the class_name.

The important point to note is the use of the scope resolution operator (:) along with the class name in the header of the function definition. Only the scope operator identifies the functions as a member of a particular class. Without this scope operator, the function definition would create an ordinary function, subject to the usual function rules of access and scope.

Example1:

```
     Class sample
     {
         Private
         Int x;
         Int y;
       Public:
         Void readData ();//Prototype declaration
         Void display (); //Prototype declaration
     };
     Void sample :: readData () //Function definition
     {
```

```
Cin>>x>>y;
}
Void sample :: display () //Function definition
{
Cout<<x<<y;
}
```

Example2:

```
Class sample
{
        Private
        Int x;
        Int y;
  Public:
      Void readData ();//Prototype declaration
      Void display (); //Prototype declaration
};
Void sample :: readData () //Function definition
{
    Cin>>x>>y;
}
Void sample :: display () //Function definition
{
    Cout<<x<<y;
}
Class sample 1
{
    Private:
            Int j;
Public: readData()
        {
                Cin>>j;
                }
}
```

In the above example both classes are defined with the same member function names while accessing these member functions are valid.

DEFINING THE ONJECT OF A CLASS

Objects and classes have been loosely throughout the preceding section. In general, a class is a user defined data type, while an object is an instance of a class template. A class provides a template, which defines the member functions and variables that are required for objects of the class type. A class must be defined prior to the class declaration.

General Syntax:

```
Class user_ defined_ name
{
Private:
        Data members;
        Member functions;
Public:
        Data members;
        Member functions;
Protected:
        Data members;
        Member functions;
};
User_ defined_ name obj1, obj2, …, objN;
```

Where obj1, obj2, …, objN are the identical class of the user_ defined_ name

Accessing a member of a class

There are two ways one can access a member of a class similar to accessing member of a struct or union construct. A data or function member of a class construct is accessed using the period (.) operator.

General Syntax:

```
Object. Data member;
Object. Function_ member;
```

Program: A program to assign data to the data members of a class and display its contents.

```
#include<iostream.h>
Class sample
{
      Public:
            Int x;      //public members are accessed directly by the objects
            Int y;
};
Void main ()
{
            Sample s;
            Cout<<"Enter x and y values "<<endl;
            Cin>>s.x>>x.y;
            Cout<<"x value is"<<s.x<<"y value is "<<endl<<s.y;
};
```

Output:

```
Enter x and y values
10
56
X value is 10
```

```
Y value is 56
```

Program: A program to assign data to the data members of a class using the function readData () and display its contents using the function display ().

```
#include<iostream.h>
Class sample
{

    Private:
    Int x;
    Int y;
    Public:
    Void readData()
{
    Cout<<"Enter x and y values"<<endl;
    Cin>>x>>y;
}
Void display ()
{
    Cout<<"x"<<x<<"y value is"<<endl<<y;
    }
};
Void main ()
{
     Sample s;
     s. readData ();
     s. display ();
}
```

Output:

```
Enter x and y values
10
56
X value is 10
Y value is 56
```

Program: A program to assign data to the data members of a class using the function readData () and display its contents using the function display (). Define the function outside its class (scope).

```
#include<iostream.h>

Class sample
{
     Private:
     Int x;
     Int y;
Public:
     Void readData();
     Void display ();
};
Void sample :: readData () //Function defined out side the class sample.
{

        Cout<<”Enter x and y values”<<endl;
        Cin>>x>>y;
}
Void sample:: display ()
{
        Cout<<”x value is”<<x<< endl<<”y value is”<<endl<<y;
}
Void main ()
{
     Sample s;
     S. readData ();
     S. display ();
}
```

Output:

```
Enter x and y values
X value is 56
Y value is
78
```

Program: A program to perform simple complex numbers addition and multiplication using an OOP technique.

```
#include<iostream.h>
Class sample
{
        Private:
        Int rp;
        Int ip;
Public:
        Void readData ();
        Void display ();
        Void add (sample s1, sample s2);
        Void sub (sample s1, sample s2);
        Void mul (sample s1, sample s2);
        Void div (sample s1, sample s2);
};
        Void sample :: readData ()
{
                Cout<<”Enter real and imaginary part values”<<endl;
                Cin>>rp>>ip;
}
Void sample :: display ()
```

```
{
            Cout<<rp<<"+"<<"I"<<endl;
}
Void sample ::add (sample s1, sample s2)
{
            Cout<<s1.rp+s2.rp<<"+"<<s1.ip+s2<<"I"<<endl;
}
Void sample ::sub(sample s1, sample s2)
{
            Cout<<s1.rp-s2.rp<<"-"<<s1.ip-s2.ip<<"I"<<endl;
}
Void sample :: mul (sample s1, sample s2)
{
        Int creal, simag;
        Creal=(s1.rp*s2.rp)-(s1-ip*s2.ip);
        Cimag=(s1.rp*s2.ip)+(s1.ip*s2.rp);
        Cout<<creal<<"+<<cimag<<"I"<<endl;
}
Void sample :: div(sample s1,sample s2)
{
        Int temp,creal, cimag;
        Temp=(s1.rp*s2.rp)+(s2.ip*s2.ip);
        Creal=((s1.rp*s2.rp)+(s1.ip*s2.rp))/temp;
        Cimag=((s1rp*s2.ip)-(s1.ip*s2.rp))/temp;
        Cout<,creal<<"+"<<cimag<<"I"<<endl;
}
Void main ()
{
    Sample s1, s2, s;
    Cout<<First Complex Number"<<endl;
    S1. readData ();
    Cout<<"Second Complex Number"<<endl;
    S2. readData ();
    Cout<<"First Complex Number is";
    S1. display ();
    Cout<,"Second Complex Number is";
    S2. display ();
    Cout<<"Addition value becomes "<<endl;
    s. add (s1, s2);
    cout<<"Subtraction value becomes"<,endl;
    s. sub(s1,s2);
    cout<<"Multiplication value becomes"<,endl;
    s. mul(s1, s2);
    cout<<"Division value becomes"<,endl;
    s.div(s1,s2);
}
```

Output:

```
First Complex Number
Enter real and imaginary part values
5.5
Second Complex Number
Enter real and imaginary part values
2 1
First Complex Number is 5+5i
Second Complex Number is 2+1i
Addition value becomes
7+6i
Subtraction value becomes
3+4i
Multiplication value becomes
5+15i
Division value becomes
1+0i
```

ARRAY OF CLASS OBJECTS

An array is a use defined data type whose member is homogeneous and stored in contigous memory locations. For practical applications such as designing a large size of database, arrays are very essential.

General Syntax:

```
        Class user_ defined_ name
        {
                Private:
                        Data members;
                        Member functions;
                Public:
                        Data members;
                        Member functions;
                Protected:
                        Data members;
                        Member functions;
};
Class user_ defined_ name object[size];
Where size is a user defined size of the array of class objects.
```

Example:

```
Class sample
{
        Private:
        Int x;
        Int y;
Public:
      Void readData()
      {
      Cin>>x>>y;
}
Void display ()
{
      Cout<<x<<y;
}

};
Sample s[100];
```

Where s is an object of the class without tag name whose size is 100.

POINTER AND CLASSES

A pointer is a variable which holds the memory address of a variable of basic data type such as int, float or sometimes an array. A pointer can be used hold the address of a class variable too. The pointer variable is very much used to construct complex data base using the data structures such as linked lists, double linked lists and binary trees.

General Syntax:

```
      [Storage_ class] struct user _defined_ name
      {
            Data_type member1;
            Data_ type member2;
            ……………………..
            ……………………..
            Data_ type memberN;
      };*ptr1;
```

Where ptr1 is a pointer variable holding the address of the class object and is its having some members.

The pointer structure variable can be accessed and processed in one of the following ways:

(*class_ name).field_ name=variable;

The parentheses are essential because the structure member period (.) has a higher precedence over the indirection operator (*). The pointer to structure can also be expressed using dash (-) followed by the greater than sign (>).

General syntax:

Class_ name-> field_ name=variable;

Example1:

```
Class student
{
        Int rollno;
        Int mark;
}*st;

*(st).rollno=10001;
*(st).mark=87;
```

Example 2:

```
#include<iostream.h>
Void main ()
{
Struct student
{
        Int rollno;
        Int mark;
};
Struct student *st;
St->rollno=10001;
St->mark=40
Cout<<st->rollno;
Cout<<st->mark;
}
```

Output:

```
10001 40
```

UNIONS AND CLASSES

A union has been defined as a user defined data type whose size is sufficient to contain one of its members. At most, one of the members can be stored in a union at any time. A union is also used for declaring classed. The members of a union are public by default.

A union allows to store its members only one at a time. A union may have member functions including constructors and destructors, but not virtual functions. A union may not have base class. An object of a class with a constructor or a destructor or a user defined assignment operator cannot be a member of a union. A union can have no static data member.

General Syntax:

```
Union user_ defined_ name
{
Private:
        Data members;
        Member functions;
Public:
        Data members;
```

```
            Member functions;
    Protected:
            Data members;
            Member functions;
    };
    User_ defined_ name obj1, obj2, …, objN;
```
Where obj1, obj2, …, objN are the union variables.

```
    #include<iostream.h>
    Class sample
    {

        Private:
        Int x;
        Int y;
    Public:
        Void readData()
    {
        Cout<<"Enter x and y values"<<endl;
        Cin>>x>>y;
    }
    Void display ()
    {
        Cout<<"x value is"<<x<<"y value is"<<endl<<y;
        }
    };
    Void main ()
    {
          Sample s;
          S. readData ();
          S. display ();
    }
```
Output:
```
    Enter x and y values
    10
    56
    X value is 10
    Y value is 56
```

CLASSES WITHIN CLASSES (NESTED CLASSES)

C++ permits declaration of a class within another class. A class declared as a member of another class is called as a nested class or a class within another class. The name of a nested class is local to the enclosing class. The nest class is in the scope of its enoclosing class.

General Syntax:

```
        Class outer_ class
        {
                Private:
                      Members;
                Public:
                      Members;
                            Class inner_ class
                            {
                                Private:
                                      Data Members;
                                      Member function;
                                Public:
                                      Data Members;
                                      Member function;
                                Protected:
                                      Data Members;
                                      Member functions;
                            };//End of Inner class
          Protected:
                  Members
    };//End of outer Class
Outer_ class object1;
Outer_ class:: inner_ class object2;
```

Note: Simply declaring a class nested in another does not mean that the enclosing class contains an object of the enclosed class. Nesting expresses scoping, not containment of such objects.

Program: A program to demonstrate the nested classes

```
      #include<iostream.h>
      Class student
      {
          Private:
                Int roll;
                Int mark;
          Public:
              Class dob
              {
                    Public:
                Int dd;
                Int mm;
                Int yy;
                Void readDOB ()
                {
                Cout<<"Enter the Date of Birth  \n";
                Cin>>dd>>mm>>yy;
                }
                Void displayDOB()
```

```
            {
            Cout<<"Marks \t";
            Cout<<dd<<"-"<<mm<<"-"<<yy;
            }
        };
        Void readStudent ()
        {
        Cout<<"Enter the roll number and mark;
        }
        Void displayStudent()
        {
        Cout<<"Student Details";
        Cout<<rollno<<"\"<<mark<<"\t";
        }
};
Void main ()
{
Student st;
St. readStudent ();
Student::dob d;
d. readDob();
st.displayStudent ();
d. displayDob();
}
```

Output:

```
        Enter the roll number and mark
        10001
        98
        Enter the Date of Birth
        10
        10
        98
        Student Details 10001 98 Marks 10- 10- 98
```

CONSTRUCTORS

A constructor is a special member function for automatic initialization of an object. Whenever an object is created, the special member function, i. e. , the constructor will be executed automatically. A constructor function is different from all other nonstatic member functions in a class because it is used to initialize the variables of whatever instance being created. Note that a constructor function can be overloaded to accommodate many different forms of initialization.

Rules for writing constructor functions

- A class name and the constructor name must be the same.

- It is declared with no return type even void also
- It cannot be declared const or volatile but a constructor can be invoked a canst and volatile objects.
- It may not be static
- It may not be virtual
- It should have public or protected access within the class and only in rare circumstances it should be declared private.

General Syntax:

```
Class user_ name
{
        Private:
              ……………………;
              ……………………;
        Protected:
              ……………………;
              ……………………;
        Public:
                User_ name (); //Constructor
                …………………;
                …………………..;
};
User_ name :: user _ name ()
{
          ..............................;
          ……………………...;
}
```

Example:

```
#include<iostream.h>
Class test
{
Private:
      Int I, j;
Public:
      Test ()
      {
              I= 0;
              J= 10;
      }
      Void display ()
      {
            Cout<<"I value "<<i<<endl;
            Cout<<"j value"<<j<<endl;
      }
};
Void main ()
{
```

```
                Test t;
                t. display ();
        }
```
Output:

```
        I value 0
        J value 10
```

A constructor is automatically invoked when an object begins to live.

TYPES OF CONSTRUCTOR

1. Default constructors
2. Copy constructors

1. Default constructors

A constructor without argument is called ***Default constructor*** and with arguments called. ***Parameterize constructor.***

General Syntax:

```
        Class user_ name
        {
            Private:
                ……………………;
                ……………………;
            Protected:
                ……………………;
                ……………………;
            Public:
                User_ name (); //Constructor
                User_ name (data_ type arg1, …, data_ type argN); //Parameterized
            constructor
                …………………;
                …………………..;
        };
        User_ name :: user _ name ()
        {
                ..............................;
                …………………...;
        }
```
Example:

```
        #include<iostream.h>
        Class test
        {
        Private:
```

```
        Int I, j;
Public:
        Test ()
        {
                I= 0;
                J= 10;
        }
        Test(int a1)
        {
                  I=a1;  //// j could not initialized so, it returns the address of the
Variable
        }
        Test(int a1, int a2)
        {
                I= a1;
                J=a2;
        }
        Void display ()
        {
              Cout<<"I value "<<i<<endl;
              Cout<<"j value"<<j<<endl;
        }
};
Void main ()
{
            Test t;
            Cout<<"output of default constructor"<,endl;
            t. display ();
            test t1 (100);
            cot<<"output of parameterized constructor with 1 parameter"<<endl;
            t1. display ();
            test t2(54, 67);
            cout<<"output of parameterized constructor with 2 parameters"<<endl;
            t2. display ();
}
```

Output

```
Output of default constructor
                  I value 0
                  J value 10
Output of parameterized constructor with 1 parameter
                  I value 100
                  J value 1164
Output of parameterized constructor with 2 parameters
                  I value 54
                  J value 67
```

2. Copy constructor

Copy constructors are always used when the compiler has to create a temporary object of a class object. The copy constructors are used in the following situations:

- The initialization of an object by another object of the same class.
- Return of objects as function value
- Stating the object as by value parameters of a function

General Syntax:

Class_ name:: class_ name(class_ name &ptr)
where class_ name is the user defined class name.

The copy constructor may be used in the following format also using a const keyword.

Class_ name:: class_ name(const class_ name &ptr)
where class_ name is the user defined class name.

program: A program to generate a series of Fibonacci numbers using a copy constructor where the copy constructor id defined with the class declaration itself.

```
#include<iostream.h>
Class fib
{
Private:
    Unsigned long int f0, f1, fib;
Public:
Fib ()
{
        F0=0;
        F1=1;
        Fib=f0+f1;
}
Fib:: fib(fib &ptr)
{
        F0=ptr. f0;
        F1=ptr. f1;
        Fib=ptr. fib;
}
Void increment ()
{
        F0=f1;
        F1=fib;
        Fib=f0+f1;
}
Void display ()
{
          Cout<<fib<<"\t";
}
};
Void main ()
```

```
{
        Fib f;
        For(int i=0;i<=10;++i)
        {
             f. display ();
             f. increment ();
}
}
```

DESTRUCTORS

A destructor is a function that automatically executes when an object is destroyed. A destructor function gets executed whenever an instance of the class to which it belongs out of existence. A primary usage of the destructor function is to release space on the heap. A destructor function may be invoked explicitly.

Rules for writing a destructor function

- A destructor function name is the same as that of the class it belongs except that the first character of the name must be a tilde (~).
- It is declared with no return types since it cannot ever return a value.
- It cannot be declared static, const or volatile.
- It takes no arguments and therefore cannot be overloaded.
- It should have public access in the class declaration.

General Syntax:

```
Class user_ name
{
          Private:
                 ……………………;
                 ……………………;
          Protected:
                 ……………………;
                 ……………………;
          Public:
                   User_ name (); //Constructor
                   User_ name (data_ type arg1, …, data_ type argN); //Parameterized
          Constructor
                   ~user_ name(); //Destructor
                   …………………;
                   ………………….;
};
~User_ name :: user _ name ()
{
            ...............................;
```

```
                …………………...;
        }
```

Example:

```
        #include<iostream.h>
        Class test
        {
        Private:
                Int I;
        Public:
                Test ()
                {
                        I= 100;
                        Cout<<"\nConstructor called  automatically "<<endl;
                }
                ~Test(int a1)
                {
                        Cout<<"\nConstructor called  automatically "<<endl;
                }
        };
        Void main ()
        {
                Test t;
        }
```

Output:

```
        Constructor called automatically
        Destructor called automatically
```

INLINE MEMBER FUNCTIONS

The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

The advantages of using inline member functions are

- The size of the object code is considerably reduced.
- It increases the execution speed
- The inline member functions are compact function calls

General Syntax:

```
        Class user_ name
        {
                Private:
                        ……………………
                        ……………………
                Public:
```

```
                              Inline return_ type function_ name(parameters);
                              Inline return_ type function_ name(parameters);
                              ……………………
                              ……………………
          }
```

The keyword inline is used as a function specifier only in function declarations. It can be used either as a member of a class or a global function. To define inline member specifier is well suited whenever a function is small, straight forward and are not called from too many different places.

Example:

```
    #include<iostream.h>
    Class sample
    {

        Private:
        Int x;
        Int y;
    Public:
        Inline void readData()
    {
        Cout<<"Enter x and y values"<<endl;
        Cin>>x>>y;
    }
    Inline void display ()
    {
        Cout<<"x value is"<<x<<endl<<"y value is"<<endl<<y;
        }
    };
    Void main ()
    {
          Sample s;
          s. readData ();
          s. display ();
    }
```

Output:

```
    Enter x and y values
    X value is 10
    Y value is 20
```

STATIC CLASS MEMBERS

Static variables are automatically initialized to zero unless it has been initialized by some other value explicitly. Static members of a class can be categorized into two types: **static data member** and **static member function.**

Whenever a data or function member is declared as a static type, it belongs to a class, not to the instances or objects of the class. Both the data member and member function can have the keyword static.

STATIC DATA MEMBER

Static data member are data objects that are common to all the objects of a class. They exist only once in all objects of this class. They are already created before the finite object of the respective class. The static member are used information that is commonly accessible. Static member can be of any one of the groups: public, private and protected, but not global data. The a class is public, it can be used as a normal variable.

A static data member of a class has the following properties

1. The access rule of the data member of a class is same for the static data member also. If a static member is declared as a private category of a class, then non member functions cannot access these members. If a static member is declared as public then any member of the class can access.
2. Whenever a static data member is declared and it has only a single copy, it will be shared by all the instance of class. That is, the static member becomes global instances of the class.
3. the static data member should be created and initialized before the main () function control block begins.

General Syntax:

```
Class user_ defined_ name
{
        Private:
              Static data_ type variables;
              Static data_ type variables;

              ……………………
              ……………………
        Public:
              ……………………
              ……………………
}
```

Example1:

```
#include<iostream.h>
Class sample
{
      Private:
      Static int x;
      Public:
      Void display ()
      {
```

```
                Cout<<"x value is "<<s<<endl;
        }
    };
    Int sample:: x;
    Void main ()
    {
        Sample s;
        s. display ();
    }
```
Output:

```
     X value is 0
```
Examjple2:

```
    #include<iostream.h>
    Class sample
    {
            Private:
            Static int x;
            Public:
            Void display ()
            {
                    Cout<<"x value is "<<x++<<endl;
            }
    };
    Int sample:: x;
    Void main ()
    {
        Sample s, s1;
        s. display ();
        s1. display(); // currently x value is 1 because static variables are globally
        //accessed
        }
```
Output:

```
        X value is 0
        X value is 1
```

STATIC MEMBER FUNCTIONS

The static function is a member function of class and the static member function can manipulate only on static data member of the class. The static member function acts as global for members of its class without affecting the rest of the program. The purpose of static member is to reduce the need for global variables by providing alternatives that are local to a class. A static member function is not part of objects of class. Static members of a global class have external linkage. A static member function does not have a this pointer so it can access nonstatic members of its class only by using. Or->.

The static member function cannot be a virtual function. A static or nonstatic member function cannot have the same name and the same arguments type. And further, it cannot be declared with the keyword const. the static member function instance dependent, it can be called directly by using the class name and the scope resolution operator. If it is declared an defined in a class, the keyword static should be used only on declaration part.

General Syntax:

```
Class user_ defined_ name
 {
        Private:
              ……………………
              ……………………
        Public:
              Static data_ type function_ name (arguments);
              ……………………
 }
```

Example:

```
#include<iostream.h>
Class sample
{
        Private:
        Static int x;
        Public:
        Sample ()
        {
              X++;
        }
        Static void display ()
        {
        Cout<<"x value is "<<x<,endl;
        }
};
Int sample ::x;
Void main ()
{
Sample:: display ();
Sample s, s1
Sample ::display ();//Static member functions are accessed without creating objects
}
```

Output:

```
X value is 0
X value is 2
```

FRIEND FUNCTIONS

The main concepts of the object oriented programming paradigm are data hiding and data encapsulation. Whenever data variable are declared in a private category of a class, these members are restricted from accessing by non-member functions: The private data values can be neither read nor written by non member functions. To solve this problem, a friend function can be declared to have access to these data members. Friend is a special mechanism for letting non-member functions access private data. A friend function may be either declared or defined within the scope of a class definition. The keyword friend informs the compiler that is not a member function of the class.

A friend declaration is valid only within or outside the class definition
General Syntax:

```
        . Class user_ defined_ name
          {
                 Private:
                       ……………………
                       ……………………
                 Public:
                       Friend return_ type function_ name (parameters);

          };
```

Example:

```
#include<iostream.h>
Class sample
{
          Private:
          Int x;
          Public:
          Sample ()
          {
                X=100;
          }
          Friend void display (sample s);
  };
          Void display (sample s)//Non member function of the class sample
          {
          Cout<<"x value is "<<s. x<<endl;
          }

Void main ()
{
Sample s;
Display (s);
}
```

Output:

```
X value is 100
```

Example:

```
#include<iostream.h>
Class sample
{
        Private:
        Int x;
        Public:
        Sample ()
        {
            X=100;
        }
        Friend void display (sample s);
  };
        Void display (sample s)//Non member function of the class sample
        {
        Cout<<"x value is "<<s. x<<endl;
        }

Void main ()
{
Sample s;
Display (s);
}
```

Output:

```
X value is 100
```

(a) Accessing private data by non-member function through friend. The private data members are available only to the particular class and not to any other part of the program. A non-member function cannot access these private data.

Each time a friend function accesses the private data, naturally the level of privacy of the data encapsulation gets reduced. Only if it is necessary to access the private data by non-member functions, then a class may have a friend function, otherwise it is not necessary.

Example: Above example

(b) friend function with inline substitution

Friend function may also have inline member functions. If the friend function is defined with the scope of the class definition, then the inline code substitution is automatically made. If it is defined outside the class definition, then it is required to precede the return type with the keyword inline in order to make a inline code substitution.

Example:

```
#include<iostream.h>
Class sample
{
        Private:
        Int x;
```

```
            Public:
            Sample ()
            {
                  X=100;
            }
            Friend void display (sample s);
  };
            Inline void display (sample s)//Non member function of the class sample
            {
            Cout<<"x value is "<<s. x<<endl;
            }

            //int sample :: x;
Void main ()
{
Sample s;
Display (s);
}
```

Output:

```
X value is 100
```

(c) Granting friendship to another class. A class can have friendship with another class.

Example:

```
 #include<iostream.h>
Class sample
{
            Friend class second;
            Private:
            Int x;
            Public:
            first ()
            {
                  X=100;
            }

  };
  Class second
  {
            Public:
            Inline void display (sample s)//Non member function of the class sample
            {
            Cout<<"x value is "<<s. x<<endl;
            }

Void main ()
{
```

```
            First f;
            Second s;
            S. display(f);
            }
```
Output:

```
            X value is 100
```

(d) Two classes having the same friend.

General Syntax:

```
              Class class1
              {
                    Private:
                    …………………
                    ………………....
              Public:
                    Friend return_ type function_ name(parameters);
              };
              Class class2
              {
                    Private:
                          …………………
                          …………………
                          Public:
                              Friend return_ type function_ name(parameters);
              };
              Return_ type function_ name(parameters)
              {
                  …………………
                  …………………
              }
```

Example:

```
            #include<iostream.h>
            Class second; //forward declaration
            Class first
            {
                      Private:
                            Int x;
                      Public:
                      first ()
                      {
                            X=100;
                      }
                      Friend void sum (first f,second s);

              };
```

```
Class second
{
        Private:
              Int y;
        Public:
              Second ()
              {
                    Y= 200;
        }

        Friend void sum (first f, second s);
  }
  Void sum (first f, second s)
  {
        Cout <<"x value is "<<f. x<<endl;
        Cout<<"y value is <<s. y<<endl;
        Cout<<"The summation of x and y value is <<"f.x +s.y;
  }
  Void main ()
  {
        First f;
        Second s;
        Sum (f. s);
  }
```

Output:

```
            X value is 100
            Y value is 200
The summation of x and y value is 300
```

DYNAMIC MEMORY ALLOCATIONS

Two operators namely, new and delete are used in dynamic memory allocations which are described in detail in this section.

New

The new operator is used to create a heap memory space for an object of a class. In C, malloc(), calloc() and alloc() functions are used to create a memory space dynamically. C++ provides a new way in which dynamic memory is allocated. The new keyword calls upon the function operator new() to obtain storage.

Basically, an allocation expression must carry out the following three things:

- Find storage for the object to be created
- Initialize that object
- Return a suitable pointer type to the object

The new operator returns a pointer to the object created. Functions cannot be allocated this way using new operator but pointers to functions be used for allocating memory space.

General Syntax:

Data_ type pointer= new data_ type;

Where data_ type can be a short, int, float, char, array or even class objects.

Example:

```
New int;        //an expression to allocate a single integer
New float;      //an expression to allocate a floating value
```

If the call to the new operator is successful, it returns a pointer to the space that is allocated. Otherwise it returns the address zero if the space could not be found or if some kind of error is detected.

Delete

The delete operator is used to destroy the variable space which has been created by using the new operator dynamically. It is analogous to the function free() in C. The keyword delete calls upon the function operator delete() to release storage which was created using the new operator.

General Syntax:

Delete pointer;

Example:

```
char*ptr_ch=new char;     //memory for a character is allocated
Int*ptr_i=new int;        //memory for an integer is allocated
Delete ptr_ch;            //delete memory space
Delete ptr_i              //delete
```

Note: delete operator is used for only releasing the heap memory space which was allocated by the new operator. if attempts are made to release memory space using delete operator that was not allocated by the new operator, then it give unpredictable results. the following usage of the delete operator is invalid, as the delete operator should not be used twice to destroy the same pointer.

Example:

```
char *ptr_ch=new char;    //memory for a character is allocated
delete ptr_ ch;           //delete memory space
delete ptr_ch;            //error because already destroy the pointer
```

program: A program to create a dynamic memory allocation for the standard data types: integer, floating point, character and double. the pointer variables are initialized with some data and the contents of the pointers are displayed on the screen.

```
#include<iostream.h>
void main()
{
    int *ptr_i=new int(25);
    float *ptr_i=new(-10.20);
    char *ptr_c=new char(‘a’);
    double *ptr_d=new double(1234.493);
    cout<<”Integer value “<<*ptr_i<<endl;
    cout<<”Float value”<<*pyr_f<<endl;
```

```
        cout<<"Character value"<<*ptr_c<<endl;
        cout<<"Double value"<<*ptr_d<<endl;
        delete ptr_i;
        delete ptr_f;
        delete ptr_c;
        delete ptr_d;
    }
```

Output:

```
        Integer value 25
        Float value- 10.2
        Character value a
        Double value 1234.493
```

Array data type when an object is an array data type, a pointer to its initial element is returned.

```
        new int;
        new int[20];
```

Both the expressions return a pointer to the first element of the array as

    int *;

General Syntax:

    data_ type pointer=new data_type[size];

where data_type can be a short, int, float, char, array or even class objects, and size is the maximum number of elements that are to be accommodated.

Example:

int *ptr_a=new int[20]; //an expression to allocate a memory space for 20 integers using new operator;

char *ptr_ch=new char[100]; //an expression to create memory space for 100 characters using new operator.

use of new operator to allocate memory for a two dimensional array A toe dimensional array can be declared using the new operator as

dimensional array can be declared using the new operator as

new int [10][20];

which returns

int (*0[20];

General Syntax:

    data_type(pointer)[size]=new data_type[size];

Example:

int (*ptr_a)[5]+new int[5][5]; //an expression to allocate memory space for 5x5 integers using new operator.

int (*ptr_c)[10]+new int [10][10]; //an expression to create memory space for 10 x 10 characters using new operator.

the following section shows how the delete operator is used to destroy the objects created by new operator for the array data type. the expression for the delete operator is same for both the one dimensional and multidimensional arrays.

General Syntax:

```
delete[] pointer;
Example:
char *ptr_ch=new char[100];
……………………………..
……………………………..
delete[]ptr_ch;
```

this POINTER

a pointer is a variable which holds the memory address of another variable. using the pointer technique, one can access the data of another variables indirectly. the this pointer is a variable which is used to access the address of the class itself. Sometimes the this pointer may have return data items to the caller.

General Syntax:

```
this -> var=Value;       //assign a value to the variable var
x=this ->var               //retrieve the value using this pointer
```

program: A program to display the object's address of a class using this pointer.

```
#include<iostream.h>
class sample
{
    private:
            int I;
    public:
            void display()
            {
            cout<<"Objects address: "<<this<<endl;
            }
    };
    void main ()
    {
    sample s1, s2, s3;
    s1.display();
    s2.display();
    s3.display();
    }
```

Output:

```
Objects address: 0x1157fff4
Objects address:0x1157fff2
Objects address:0x1157fff0
```

Program A program to demonstrate how the this pointer is used to access the member data of a class.

```
#include<iostream.h>
class sample
{
        private:
                int I;
        public:
                sample(int i)
                {
                      this-> i=I;             //assign value using this pointer
                }
                void display()
                {
                          cout<<"Object Value:"<<this->i<<endl;//access value using this pointer
                }
        };
        void main()
        {
              sample s1(17),s2(123),s3(10);
              s1.display();
              s2.display();
              s3.display();
        }
```

output:

```
        Object Value: 17
        Object Value: 123
        Object Value:10
```

INHERITANCE

Inheritance is the processes of creating new classes from an existing class. the existing class is known as the base class and the newly created class is called as a derived class. the derived class inherits some or all of the trait from the base class. it can also add some more features to class. The base class is unchanged by its process.

Advantages

- Reusability of the code
- To increase the reliability of the code and
- To add some enhancements to the base class.

Once the base class is written and debugged, it need not changed again when there are circumstances to add or modify the member of the class.

Defining the derived class

A derived class is defined by specifying its relationship with the base class in addition to its own details.

General Syntax:

Class derived_ class_ name: visibility_ mode base_ class_ name

```
{
        Members of derived class
        …………………………
        …………………………
}
```

The colon indicates that the derived_ class_ name is derived from the base_ class_ name. The visibility mode is optional and, if present, may be either private or public. The default visibility_ mode is private. Visibility mode specifiers whether the features of the base class are privately derived or publicly derived.

Example:

```
Class base: private xyz //private derivation
{
          Member of base
          ………………
          ………………
};
Class der: public base //derivation
          Member of base
          Member of der
          ………………
          ………………
};
```

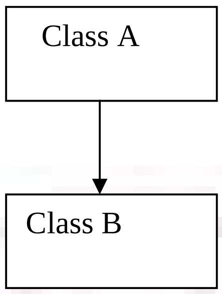
When a base class id privately inherited by a derived class, ‘public members’ of the base class become ‘private members’ of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class an be accessed its own objects using the dot (.) operator. The result is that no member of the base class is accessible to the objects of the derived class.

When the base class is publicly inherited, ‘public members’ of the base class become ‘public members’ of the derived class and therefore they are accessible to the objects of the derived class. In both cases, the private members are not inherited and therefore the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are ‘inherited’ into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

SINGLE INHERITANCE

Single inheritance is process of creating new classes from an existing base class. The existing class is known as the direct base class and the newly created class is called as a singley derived class. Single Inheritance is the ability of a derived class to inherit the member functions and variables of the existing base class.



Example:

```
#include<iostream.h>
class base
{
    private: int k;
    public: int I, j;
            void read ()
            {
                  cin>>I;
                  cin>>j;
                  cin>>k;
            }
            void display ()
            {
                  cout<<"i :"<< i<<end<<l<<"j :"<< j<<endl<<"k :"<< k<<endl;
            }
};
class der: public base
{
    public:
    void sum ()
    {
          cout<<"I and j Summation is "<<i+j+k;
          // I and j are inherited publicly, k not accessible here
    }
};
void main ()
{
  der d;
  d. read ();
  d. display ();
  d.sum ();
```

Output:

```
i: 10
j: 20
k: 30
I and j Summation is 30
```

TYPES OF BASE CLASSES

Any class can serve as a base class. A derived class may be defined as a base of another class. A base class can be classified into two types, direct base and indirect base.

Direct base class

A base class is called a direct base if it is mentioned in base list.

Example:

```
Class base
{
…………
………...
};
class der: public base
{
…………
…………
};
```

Indirect base class

A derived class can itself serve as a base class subject to access control. When a derived class is declared as a base of another class, the newly derived class inherits the properties of its base classes including its data members and member functions. A class is called as an indirect base if it is not a direct base, but it’s base class of one of the classes mentioned in the base list.

Example:

```
Class base
{
…………
………...
};
class der1: public base
{
…………
…………
}
class der2: public der1
{
  ………..     //indirectly inherited the members of the class 'base'
  };
};
```

TYPES OF DEPIVATION

Inheritance is a process of creating a new class from an existing class. while deriving the new classes, the access control specifier gives the total control over the data members and methods of the base classes. A derived class can be defined with one of the access specifiers: private, public and protected.

Public Inheritance

- Each public member in the base class is public in the derived class
- Each protected member in the base class is protected in the derived class
- Each member in the base class remain private in the base class.

General Syntax:

```
Class base
{
………..
………...
};
class der: public base
{
………..
………..
};
```

Private Inheritance

- Each public member in the base class is private in the derived class.
- Each protected member in the base class is private in the derived class.
- Each private member in the base class remains private in the bare class and hence it is visible only in the base class.

General Syntax:

```
Class base
{
………..
………...
};
class der: private base
{
………..
………..
};
```

Protected Inheritance

- Each public member in the base class is private in the derived class.
- Each protected member in the base class is private in the derived class.
- Each private member in the base class remains private in the bare class and hence it is visible only in the base class.

General Syntax:

```
Class base
{
…………
………...
};
class der: protected base
{
…………
…………
};
```

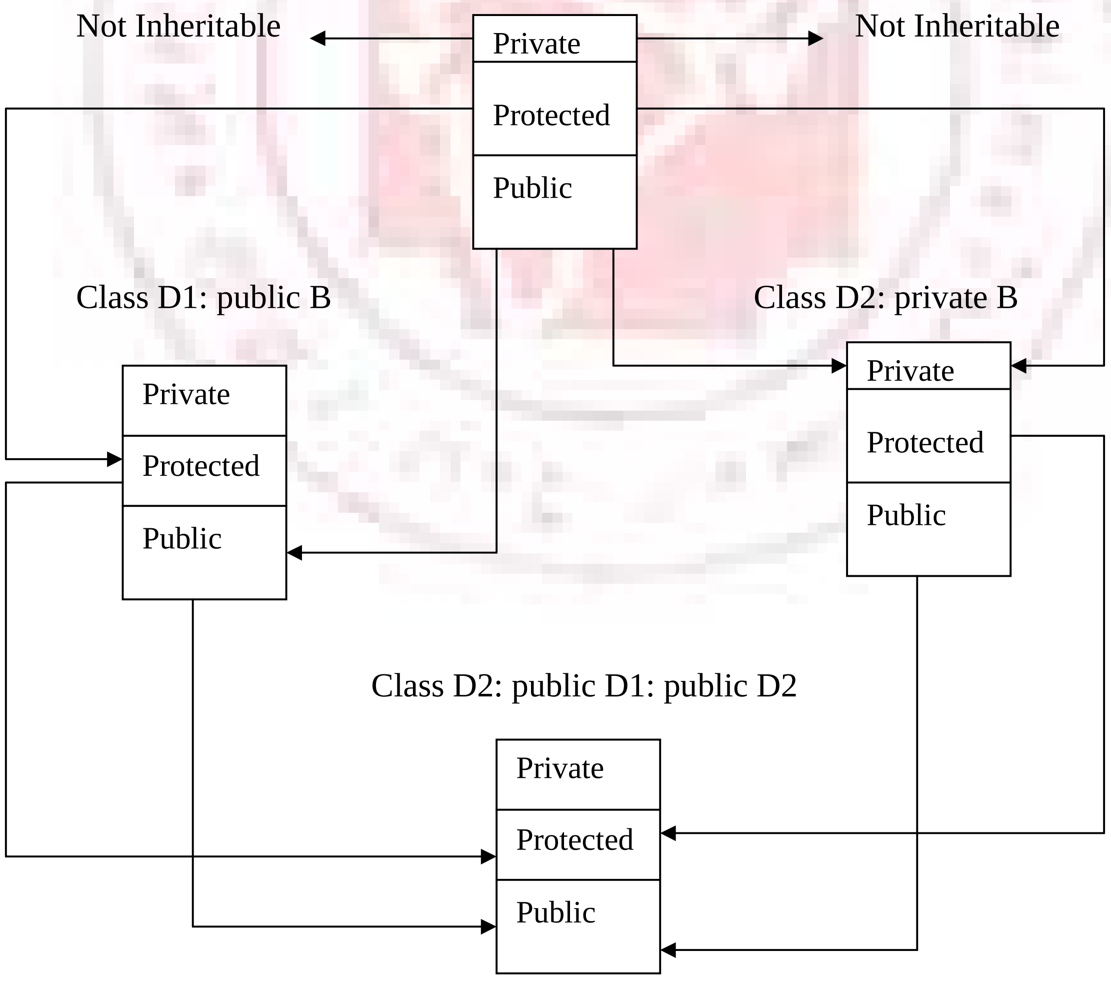


Fig. Effect of inheritance on the visibility of members

AMBIGUITY IN SINGLE INHERITANCE

Whenever a data member and member function are defined with the same name in both the base and the derived classes, these names must be without ambiguity. The scop resolution operator (::) may be used to refer to base member explicitly. This allows access to a name that has been redefined in the derived class.

Example:

```
        #include<iostream.h>
        class base
        {
            public:
                    void display ()
                    {
                          cout<<"Base class"<<endl;
                    }
};
class der:public base
{
            public:
                    void display ()
                    {
                          cout<<Derived class"<<endl;
                    }
};
void main()
{
        der d;
        d. display();
        d. base:: display();
}
```

Output:

```
Base class
Derived class
```

ARRAY OF CLASS OBJECTS AND SINGLE INHERITANCE

Once a derived class has been defined, the way of accessing a class member of the array of class objects are same as the ordinary class types:

General Syntax:

```
Class base
{
        Private:
                …………
                …………
        Public:
                …………
                …………
};
class der : public base
{
        Private:
                …………..
                …………
        Public:
                …………
                …………
};
void main ()
{
        der obj[100]; //array of class objects of the derived class.
}
```

Program: A program to demonstrate array of class objects of a derived class

```
        #include<iostream.h>
        class student
        {
                Private:
                        int rollno;
                        char name[25];
                public:
                        void readStudent ()
                        {
                                cout<<"\nEnter the student details\n";
                                cout<<"Enter Roll No";
                                cin>>rollno;
                                cout<,"Enter name";
                                cin>>name;
        }
        void displayStudent ()
        {
                                cout<<rollno<<"\t";
                                cout<<name<<"\t\t";
```

```
};
class mark : public student
{
          private:
                    int mark;
          Public:
void readMark ()
{
                           cout<<”Enter the mark\n”;
                           cin>>mark;
}
void display Mark()
{
                           cout<<mark<,”\t”<<endl;
          }
};
void main()
{
       mark d[3];
       int I;
       for(i=0;i<3;i++)
       {
                d[i].readStudent ();
                d[i].readMark ();
       }
       cout<<”\nStudent details\n”;
       cout<<”Roll No.”<,”\t”<<”Name”<<”\t<<”<,”Mark”<<endl;
       for(i=0;<3;i++)
       {
                d[i].displayStudent ();
                d[i]. display Mark ();
       }
}
```

Output:

```
Enter the student details
Enter Roll No: 1001
Enter name: Radhakrishnan
Enter the mark 90
Enter the Student details
Enter Roll No: 10002
Enter name: jthasvi
Enter the mark 65
Enter the student details
Enter Roll No: 10003
Enter name: Subramanian
Enter the mark 87
Student details

Roll No.        Name              Mark
10001       Radhakrishnan      90
10002       jthasvi             65
10003       Subramanian         87
```

MULTILEVEL INHERITANCE

A derived class can have an indirect base class called Multi level inheritance. When a derived class is declared as a base of another class, the newly derived class inherits the properties of its base classes including its data members and member functions. A class is called as an indirect base if it is not a direct base, but it's a base class of one of the classes mentioned in the base list.

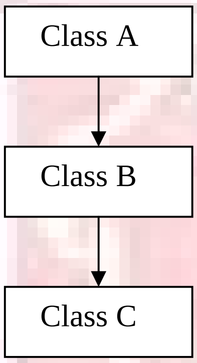
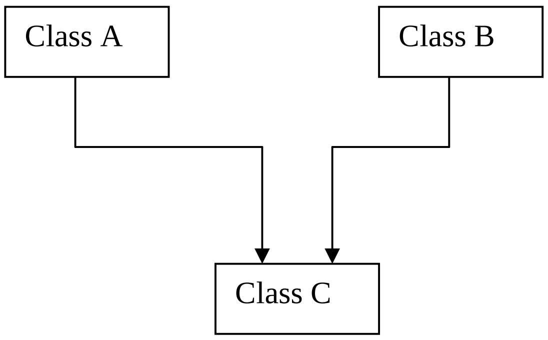


Fig. Multi Level Inheritance

MULTIPLE INHERITANCES

Multiple inheritances is the process of creating a new class from more than one base classes. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another. The syntax for inheritance is similar to that of single inheritance but the base classes are separated by commas.



General Syntax:

```
Class D: visibility Base A, visibility Base B …
{
        …………….........
        Member of class D
        …………………..
};
```

Program: A program to demonstrate Multiple Inheritance

```
      #include<iostream.h>
      Class student
      {
          Private:
                  Int rollno;
                  Char name [25];
          Public:
                  Void readStudent ()
                  {
                          cout<<"\nEnter the student details\n";
                          cout<<"Enter Roll No";
                          Cin>>rollno;
                          cout<,"Enter name";
                          Cin>>name;
                  }
                  Void displayStudent ()
                  {
                          cout<<rollno<<"\t";
                          cout<<name<<"\t\t";
                  };
                  Class mark: public student
                  {
          Private:
                  Int mark;
          Public:
                  Void read Mark ()
                  {
                        cout<<"Enter the mark\n";
                        Cin>>mark;
                  }
                  Void display Mark ()
                  {
                          cout<<mark<,"\t"<<endl;
                          }
```

```
};
Class overall: public student, public mark
{
};
Void main ()
{
        Overall d;
        d. read Student ();
        d. read Mark ();
        d. displayStudent ();
        d.Mark ();
}
```

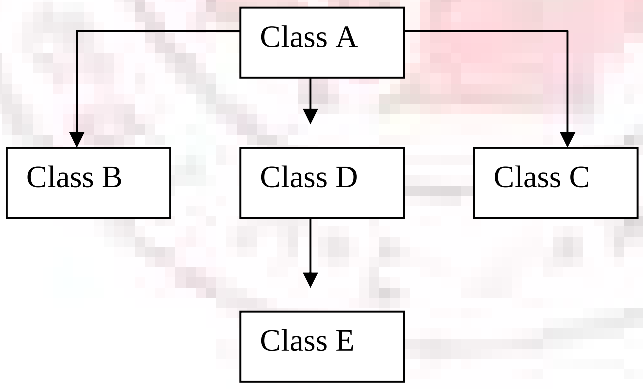
Output:

```
Enter the student details
Enter Roll No: 10001
Enter the Student details
Enter the mark 98
10001        Subramanian      98
```

Note: Ambiguity in the Multiple Inheritance is very similar to Single Inheritance ambiguity.

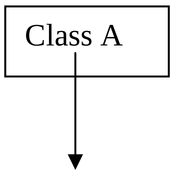
HERARCHICAL INHERITANCE

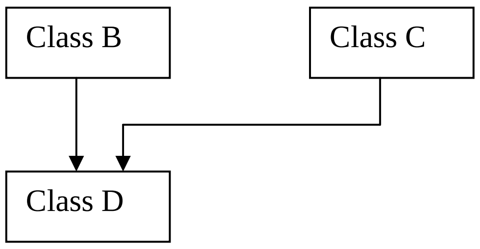
The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.



HYCRID INHERITANCE

The combination of Multilevel and Multiple inheritance is called Hybrid inheritance.





CONTAINER CLASSES

When a class is declared and defined as a member of another class, it is known as a nested class. An object of a class as a member of another class called container class.

General Syntax:

```
Class user _defined_ name1
{
        …………………..
        …………………..
};
Class user _defined_ name2
{
        …………………..
        …………………..
};
Class user _defined_ name2
{
        user_ defined_ name1 udn1; //object of the class1
        user_ defined_ name1 udn2; //object of the class2
        …………………..
        …………………..
};
```

Program: A program to demonstrate container class

```
      #include<iostream.h>
      class classA
      {
            private: int x;
                        float y;
            public:
                    void read (int a, float b)
                    {
                          x=a;y=b;
            }
            void display ()
            {
                        cout<<"x value"<<x<<"\ty value"<<y<<"\t";
            }
      };
      class classB
      {
            private: int z;
            public:
            void read (int c)
            {
                        z=c;
            }
            void display ()
            {
                cout<<"z value"<<z;
            }
      };
      class classC
      {
      private:
            classA a;
            classB b;
      public:
            void call ()
            {
                        a.read (56, 19.4);
                        b. read (45);
                        a. display();
                        b.display();
            }
      };
      void main ()
      {
            class Cc;
            c. call();
      }
```

Output:

```
    X value 56 y value 19.4  z value 45
```

MEMBER ACCESS CONTROL

The access mechanism of the individual members of a class is based on the use of the Keywords public and protected

Accessing the public data

- All the member functions of the class

- Non member functions of the class
- Member function of a friend class
- Member function of a derived class if it has been derived publicly

Accessing the private data

- Member functions of the class
- Member function of a friend class in which it is declared

Accessing the protected data

- Member functions of the class
- Member function of a friend class in which it is declared
- Member function of a derived class irrespective of whether the derived class has been derived privately or publicly

OVERLOADING

Overloading refers to the use of the same thing for different purposes. Overloading of functions and operators can be declared, defined and called in a use defined program.

FUNCTION OVERLOADING

Function overloading is a logical method of calling several functions with different arguments and data types that perform basically identical things by the same name. It is also called Functional Polymorphism.

The main advantages of using function overloading are

- Eliminating the use of different function names for the same operation
- Helps to understand, debug and grap easily.
- Easy maintainability of the code
- Better understanding of the relation between the program and the outside world.

The compiler classifies the overloaded function by its name and the number and type of arguments in the function declaration. The function declaration and definition is essential for each function with the same function name but with different arguments and data types.

General Syntax:

```
Return_ type function_ name (data_ type arg1, data_ type arg2, …, data_ type argN)
{
        …………………..
        ………………….
}

return_ type function_ name (data_ type arg1, data_ type arg2, …, data_ type argN)
{
```

```
                ……………………..
                …………………….
}
```
where function_ name are same but it contains different arguments.

Program: A program to demonstrate function overloading.

```
        #include<iostream.h>
        float area(float a, float b); //Area of the Rectangle
        void main ()
        {
                cout<<”Area of the Square is “<<area(6.7)<<endl;
                cout<<”Area of the Rectangle is”<<area(8,7)<<endl;
        }
        float area(float a)
        {
             return a*a;
        }
        float area(float a, float b)
        {
             return a*b;
        }
```

Output:

```
Area of the Square is 44.889996
Area of the Rectangle is 56
```

Scoping rules for Function Overloading

Overloading is a process of defining the same function name to carry out similar type of activities with various data items or with different arguments. The overloading mechanism is acceptable only within the same scope of the function declaration. Sometimes, one can declare the same function name for different scopes of the classes or with global and local declaration, but it does not come under the technique of function overloading.

Example:

```
        #include<iostream.h>
        class first
        {
             private:
                       int x;
             public:
                       void display();
        };
        class second
        {
             private:
                       int y;
             public:
                       void display();
        };
        void first:: display()
        {
             cout<<"Class First"<<endl;
        }
        void second:: display()
        {
             cout<<"Class Second"<<endl;
        }
        void display ()//Non member function
        {
             cout<<"Non Member function"<<endl;
        }
        void main ()
        {
             first f;
             second s;
             f. display();
             s. display();
             display();
        }
```

Output:

```
    Class First
    Class Second
    Non Member function
```

Special Features of Function Overloading

Function overloading is the process of defining two or more functions with the same name, which differ only by return type and parameters. Some of the special features of the function overloading are discussed in this section

case1: The function arguments must be sufficiently different since the compiler cannot distinguish which functions to be called when and where.

Example:

```
#include<iostream.h>
void main ()
{
        int funct 1(int);
        int funct 1(int &a);
        int x;
        …………………
        …………………
        funct 1(x);
}
```

```
int funct1 (int &a)
{
//      Error, both the arguments are same in and int&
}
```

The above function arguments are not sufficiently different unabling the compiler to distinguish between these functions, and hence displays an error message.

Case 2: When typedef is used for declaring a user defined name forfunctions and variables, it is not separate type but only a synonym for another type.

```
void main()
{
        typedef int integer
        int funct 1(int);
        int funct1(integer);
        int x;
        ………………….
        …………………
        funct1(x);
}
int funct1(int)
{
        …………………
        …………………
}
int funct1(integer)
{
//         Error, both the arguments are same
}
```

Case3: Even though the values of enumerated data types are integers, they are distinguished from the standard data type of int. so, whenever a function is declared with a function argument of int and an enumerated data type, it is valid in C++ for function overloading.

Example:

```
Void main ()
{
        enum day (mon, tue wed);
        void funct1(int i);
        void funct1(day);
        ………………..
        ………………..
}
void funct1(int t)
{
        ……………….
        ………………..
}
void funct 1(day d)
```

```
{
        ………………
        ………………
}
```

Case5: The pointer arguments of pointer variable and an array type are identical.

```
void main ()
{
          int funct1 (char*);
          int funct1 char[j];
          int x;
          ………………..
          …………………
          funct1(x);
}
int funct1(char*a)
{
          ………………….
          ………………….
}
int funct1(char a[1]    // Error
}
{
```

OPERATOR OVERLOADING

Operator overloading is the process of defining the standard operators for one or more objects. It means that-a special kind of function. Operator overloading can be carried out by means of either member functions or friend functions.

General Syntax:

return_ type operator operator_ to_ be_ overloaded (parameters);

The keyword operator must be preceded by the return type of a function which gives information to the compiler that overloading of operator is to be carried out. Only those operators that are predefined in the C++ compiler are allowed to be overloaded.

Example:

```
Void operator++();              //Unary Operator
Void operator+(int x, int y)    //Binary Operator
```

Rules for overloading operators

1. Only operators are predefined in a C++ compiler can used Users cannot create new operators such as S, @, etc.
2. Users cannot change operator templates. Each operator in C++ comes with its own template which defines certain aspects of its use, such as whether it is a binary

operator or a unary operator and its order of precedence. This template is fixed and cannot be altered by overloading. During overloading, the prefixed incrementer/decrementer and the postfix incrementer/decrementer are not distinguished.

Example:

```
++operator ()
operator++ ()
```

there is no difference between writing either prefix incrementer or postfix incrementer.

3. Overloading an operator never gives a meaning which is radically different from its natural meaning.

Example: operator *() may perform addition but the code becomes unreadable.

4. Unary operator is overloaded by means of a member function take no explicit arguments and return no explicit values. When they are overloaded by means of a friend function, they take no reference argument, namely, the name of the relevant class.

Example : for Assignment operator overloading.

```
#include<iostream.h>
class smple
{
    private:
              int x, y;
    public:
              sample()
              {
              x=0;
              y=0;
    }
    sample(int a, int b)
    {
              x=a;
              y=b;
    }
    void operator =(sample s)
    void display ();
};
void sample:: operator = (sample s)
{
              x=s.x;
              y=s.y;
}
void sample: display ()
{
      cout<<"x value"<<x<<<<endl;
      cout<<"y value"<<y<<endl;
}
```

```
void main ()
{
  sample s1(10, 20), s2;
  cout<<"Object 1 value"<<endl;
  s1. display ();
  s2=s1;
  cout<<"Object 2 value"<<endl;
  s2. display ();
}
```

Output:

```
Object 1 value
x value 10
y value 20
Object 2 value
x value 10
y value 20
```

OVERLOADING OF BINARY OPERATORS

Binary operators overloaded by means of member functions take one format argument which is the value to the right of the operator. Binary operators, overloaded by means of friend functions take two arguments.

Overloading Arithmetic Operators

Arithmetic operators are binary operators, they require two operands to perform the operation. Whenever an arithmetic operator is used for overloading, the operator overloading function is invoked with single class objects.

Program: A program to program to perform overloading of a + operator for finding the sum of the two complex numbers.

```
#include<iostream.h>
class sample
{
    private:
            int rp, ip; //Real and Imaginary values
    public:
          sample ()
          {
          x= 0;
          y=0;
          }
      sample (int a, int b)
      {
```

```
                rp=a;
                ip=b;
            }
            sample operator+(sample);
            void display ();
    };
    sample sample :: operator+(sample s)
    {
        sample t;
        t. rp=x+s.rp;
        t.ip=y+s.ip;
        return t;
    }
    void sample:; display ()
    {
            cout<,s<<"+"<<y<<"I"<<endl;
  }
  void main ()
  {
        Sample s1(10,20)s2(30,40), s3;
        cout<<"First Number"<<endl;
        s1. display ();
        cout<<"SecondNumber"<<endl;
        s2. display ();
        s3=s1+s2;
        cout<<Resultant Number"<<endl;
        s3. display ();
  }
```

Output:

```
First Number
10+20i
Second Number
30+40i
Result Number
40+60i
```

Overloading of Comparison Operators

Comparison and logical operators are binary operators that require two objects to be compared and hence the result will be one of these.

Less than
Less than or equal to
Greater than
Greater than or equal to
Equal to
Not equal to

The return value of the operator function is an integer. Operator overloading accepts an object on its right as a parameter and the object on the left is passed by this pointer.

```
#include<iostream.h>
class sample
{
	private:
			Int x;
	public:
			sample()
			{
			x=0;
			}
		sample(int a)
		{
			x=a;
		}
			int operator<(sample);
			void display();
};
int sample::operator<sample s)
{
    if (x<s.x)
    return 1;
    Else
    Return 0;
}
void sample: display ()
{
	cout<<"x value"<<x<,endl;
}
void main ()
{
	sample s1 (10),s2(30);
	cout<<"First Object"<<endl;
	s1.display ();
	cout<<"Second Object"<<endl;
	s2.display ();
	cout<<The result is "<<(s1<s2);
}
```

Output:

```
	First Object
	s value 10
	Second Object
	x value 30
```

The result is 1

OVERLOADING OF UNARY OPERATORS

Unary operators overloaded by member functions take no formal arguments, whereas when they are overloaded by friend functions they take a single argument.

Overloading of incrementer and decrementer

This operator is used to increment or decrement the object value by 1. These operators can be used as either prefix or postfix. In general, overloading of these operators cannot be distinguished between prefix or postfix operation. However, whenever a postfix operation is overloaded, it takes a single argument along with a member function of a class object.

Program: A program to demonstrate Unary Operator

```
#include<iostream.h>
class sample
{
        private:
                        int x;
        public:
                        sample ()
                        {
                        x=0;

                          }
                        sample(int a)
                        {
                        x=a;
                        }
                        int operator++ ();
                        void display ();
};
int sample :: operator ++()
{
    return x++;
}
void sample :;display ()
{
                        cout<<"x value"<<x<<endl;
}
void main ()
{
        Sample s(10);
        cout<<"Object Value"<<endl;
        x. display ();
        cout<,"Result"<<endl;
        cout<<x++;
    }
```

Output:

```
Object Value
X value 10
Result
10
```

Post fix incremented

Unlike overloading a prefix operator, overloading a postfix operation always takes a single argument so as to distinguish bet6ween the prefix and the postfix operations.

Example:

```
Fibonacci Fibonacci:: operator++ (int x);
```

The following operators can be overloaded are given below

Binary Operators

[], () new, delete, *, /, %, +, -, <<, >>, <, <=, >, >=, ==, &, ^, !=, |,&&, ||, =, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=

Unary Operators

->, !, *, &, -, ->*, +, ++, --, -

Operators common to unary and binary forms

+, -, *, &

Operators can not be overloaded

., .*, ::, ?:, sizeof, #.

POLYMORPHISM

Polymorphism is the process of defining a number of objects of different classes into a group and call the methods to carry out the operation of the object using different function calls. In other words. polymorphism means 'to carry out different processing steps by functions having same messages. It treats objects of related classes in a generic manner. The keyword virtual is used to perform the polymorphism concept. Polymorphism refers to the run time binding to a Pointer to a method.

EARLY BINDING

Choosing a function in normal way, during compilation time is called as early binding or static binding or static linkage. During compilation time, the compiler determines which function is used based on the parameters passed to the function or the function' s return type. The complier then substitutes the correct function for each invocation. Such compiler based substitutions are called static linkage. Whatever functions discussed so far in the earlier chapters, are based on static binding only.

Function calls are faster in this case because all the information necessary to call the function are hardcode.

Program: A program to demonstrate the operation of the static binding.

```
#include <iostream.h>
class first
{
        private:
                int x;
        public:
                void display ();
};
class second : public first
{
        private:
                int y;
        Public:
                 void display ();
};
void first :: display ()
{
        cout<<"First Function"<<endl;
}
void second :: display ()
{
        cout<<"Second Function"<<endl;
}
void main ()
{
        first f;
        second s;
         first *ptr;
        ptr=&s;
        ptr->display ();
        s. display ();
}
```

Output:

```
First Function
Second Function
```

The derived class second is inherited from the vase class square through public derivation. It is known that an object of a derived class not only inherits characteristics that are specific to the derived class.

POLYMORPHISM WITH POINTERS

Pointers are also central to polymorphism in C++. To enable polymorphism, C++ allows a pointer in a base class to point to either a base class object or to any derived class object. The following program segment illustrates how a pointer is assigned to point to the object of the derived class.

General Syntax:

```
Class base
{
………………
………………
}
Class der: public base
{
……………..
……………..
};
void main ()
{
        base*ptr;          //pointer to base
        der d;
        ptr=&d;            //Indirect reference d to the pointer

        …………….
        …………….
}
```

The pointer ptr points to an object of the derived class d.

By contrast, a pointer to a derived class object may not point to a base class object without explicit casting.

Note:

1. A base class pointer can point to the object of the same class or a derived class.
2. A derived class pointer cannot point to an object of a base class but it can point to the same class object.

VIRTUAL FUNCTIONS

Virtual functions let derived classes provide different versions of a base class function. You can declare a virtual function in a base class, then redefine it in any derived class, even if the number and type of arguments are the same. The redefined function overrides the base class function of the same name. Virtual functions can only be member functions.

General Syntax:

```
     Class user_ defined_ name
     {
               Private:
                         …………..
                         …………..
               Public:

                         Virtual return_ type function_ name 1 (arguments);
                         Virtual return_ type function_ name 1 (arguments);
                         Virtual return_ type function_ name 1 (arguments);
     };
```

The keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual should be preceded by a return type of the function name.

Example:

```
        Class sample
        {
              Private:
                      Int x, y;
              Public:
                      Virtual void display ();
                      Virtual int sum ();
        }
```

Note:

1. The keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.
2. A virtual function cannot be a static member because a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.
3. A virtual function cannot have a constructor member function but it can have the destructor member function.
4. A destructor member function does not take any argument and no return type can be specified for it not even void.
5. An error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.

LATE BINDING

Choosing function during execution time is called late binding or dynamic binding or dynamic linkage. Late binding reqires some overhead but provides increased power and flexibility. The late binding is implemented through virtual functions. An object of a class must be declared either as a pointer to a class or a reference to a class.

General Syntax:

```
Class base
{
        virtual void display ();
        int sum ();
        ……………
};
Class der: public base
{
        void display ();
        int sum ();
        ……………
};
void main ()
{
        base *ptr;              //pointer to base
        der d;
        ptr=&d;                 //Indirect reference d to the pointer
        ………….
        ptr-> display ()        //Run Time binding
        ptr->sum ();             Compile Time binding
}
```

Program: A program to demonstrate the run time binding of the member functions of a class.

```
#include <iostream.h>
class base A
{
        public:
                virtual void display ()
                {
                        cout<<”Base A “<<endl;
                }
};
class der A: public base A
{
        public:
                virtual void display()
                {
                        cout<<”Derived A” <,endl;
                }
};
class derB :public der A
```

```
{
        public:
                virtual void display ()
                {
                        cout<,Derived B”<<endl;
                }
};
void main ()
{
        baseA bA;
        derA dA;
        derB dB;
        baseA *ptr;
        ptr=&bA;
        ptr->display();
        ptr=&dA;
        ptr->display();
        ptr=&dB;
        ptr->display();
}
```

Output:

```
Base A
Derived A
Derived B
```

Virtual function with inline code substitution

Though virtual functions can be declared as an inline code, being the run time binding of the compiler, the inline code does not affect much of the programming efficiency. For inline code substitution, the compiler must get information about the functions, like from where they have to be invoked etc. These must be defined during the compilation time.

General Syntax:

```
Class base
{
          public:
                  Virtual inline return_ type display ();
                  Virtual inline return_ type sum ();
                  …………..
};
void main ()
{
        base b;
        b-> display ();
        b-> sum ();
}
```

PURE VIRTUAL FUNCTIONS

A pure virtual function is a type of function which has only a function declaration. It does not have the function definition.

General Syntax:

```
Class base
{
        Public:
                Virtual return_ type function_ name 1 ();
                virtual return_ type function_ name 2 ();
……………
};
```

Program: A program to demonstrate pure virtual functions.

```
#include <iostream.h>
class baseA
{
      Public:
              virtual void display()
              {
              }
};
class derA : public baseA
{
      public:
              void display ()
              {
                        cout<<”Derived A “<<endl;
              }
};
void main ()
{
baseA *ptr;
derA dA;
ptr=&dA;
ptr->display ();
}
```

Output:

```
Derived A
```

Note: A pure virtual function can also have the following format, when a virtual function is declared within the class declaration itself. The virtual function may be equated to zero if it does not have a function definition.

General Syntax:

```
Class base
```

```
{
        Public:
                virtual return-type function_ name 1() =0;
                virtual return_ type function_ name 2 () =0;
                ……………
};
```

Program: A program to demonstrate pure virtual function and equate is to zero

```
#include<iostream.h>
class baseA
{
   public:
            virtual void display()
            {
            }
};
class derA : public baseA
{
   public:
            void display ()
            {
                  cout<<"Derived A"<<endl;
            }
};
void main()
{
baseA*ptr;
derA dA;
ptr=&dA;
ptr->display();
}
```

Output:

```
Derived A
```

When an object of the derived class tries to access through the pointer of the base class members, the function invoking message will reach only the derived class members but not to the base class members as the base class member function may not have a function definition.

ABSTRACT BASE CLASS

A class which consists of pure virtual functions is called an abstract base class. In the previous section it has been discussed that a function may be defined without any statement or the function declaration may be equated to zero of it does not have the function definition part.

General Syntax:

```
Class base
```

```
{
        public:
                virtual return_ type function_ name 1() =0;
                virtual return_ type function_ name 2() =0;
};
class der : public base
{
        public:
                return_ type function_ name1();
                return_ type function_ name2 ();
}
```

Program: A program to demonstrate Abstract classes

```
#include<iostream.h>
class student
{
        public:
                virtual void read()=0;
                virtual void display()=0;
};
class stu_ details : public student
{
        Private:
                Int rollno;
                int mark;
        public:
                void red();
                void display ();
};
void stu_ details :; display()
{
    cout<<"Student details …"<<endl;
    cout<<"Roll Number. "<<rollno"\Mark"<<mark;
}
void stud_ details ::read()
{
    cout<<"Enter the Student details …"<<endl;
    cin>>rollno>>mark;
}
void main()
{
    student *ptr;
    stud_ details st;
    ptr=&st;
    ptr->read();
    ptr->display();
}
```

Output:

```
        Enter the Student details …
        Student details …
        Roll Number. 10001        Mark 67
```

CONSRTUCTORS UNDER INHERITANCE

Whenever an object of a class is created, a constructor member function is invoked automatically and when an object of the derived class is created, the constructor for that object is called. This is due to object of the derived class which contains the members of the base class also. Since the base class is also part of the derived class, it is not logical to call the constructors of the base class.

General Syntax:

```
Class class1
{
        Private:
              ……………….;
              ……………….;
        Public:
              Class 1(); //Constructor
              ……………….;
              ……………….;
};
Class class2: public class1
{
        Private:
              ……………….;
              ……………….;
        Public:
              Class 2();   //Constructor
              ……………….;
              ………………..;
};
```

Program: A program to demonstrate the constructors under inheritance.

```
#include<iostream.h>
class  base
{
      public: base()
              {
                      cout<<"Base Constructor called \n";
              }
};
class der: public base
{
        public: der()
                {
                        cout<<"Derived Constructor called\n";
                }
};
void main()
{
        der d;
}
```

Output:

```
Base Constructor called
Derived Constructor called
```

DESTRUCTOR UNDER INHERITANCE

Destructor is a special member function; it is invoked automatically to free the memory space which was allocated by the constructor function. Whenever an object of the class is getting destroyed, the destructors are used to free the heap area so that the free memory space may be used subsequently. Destructors in an inheritance hierarchy fire from derived class to a base class order.

General Syntax:

```
Class class1
{
        Private:
                ………………….;
                ………………….;
        Public:
                class1 ();  //Constructor
                ~class1 ();  //Destructor
                ………………….;
                …………………..;
};
Class class2: public class1
{
        Private:
                ……………….;
                ……………….;
                Public:
                class1 ();  //Constructor
                ~class1 ();  //Destructor
                ………………….;
                …………………..;
};
```

Program: A program to demonstrate Destructors

```
#include<iostream.h>
class  base
{
public: base()
    {
        cout<<"Base Constructor called \n";
    }
    ~base ()
    {
        cout<<"Base Destructor called\n";
    }
};
class der: public base
{
    public: der()
    {
    cout<<"Derived Constructor called\n";
    }
    ~der()
    {
        cout<<"Derived Destructor called\n";
    }
};
    void main()
    {
    der d;
    }
```

Output:

```
      Base Constructor called
      Derived Constructor called
      Derived Destructor called
      Base Destructor called
```

VIRTUAL DESTRUCTORS

We know that the destructor member function is invoked to free the memory storage by the C++ compiler automatically. But the destructor member function of the derived class is not invoked to free the memory storage which was allocated by the constructor member function of the derived class. It is because the destructor member functions are non virtual and the message will not reach the destructor member functions under late binding. So it is better to have a destructor member function as virtual and the virtual destructors are essential in a program to free the memory space effectively under late binding method.

Example:

```
    #include<iostream.h>
    class  base
    {
        public: base()   //Constructor cannot have virtual
        {
            cout<<”Base Constructor called \n”;
        }
        Virtual ~base ()
        {
        cout<<”Base Destructor called\n”;
        }
};
Class der: public base
{
        public: der()
        {
        cout<<”Derived Constructor called\n”;
        }
        ~der ()
        {
        cout<<”Derived Destructor called\n”;
        }
};
void main()
{
base *b=new der;
//delete b;
}
```

Output:

```
Base Constructor called
Derived Constructor called
```

Note the whenever instances are created at run time on the heap through the new operator, constructor member function are called automatically. When the delete operator is used, the destructors are automatically called to release the space occupied by thee instance itself. As a derived class instance always contains a base class instance, it is necessary to invoke destructors of both the classes in order to ensure that all the space on the heap is released.

VIRUTAL BASE CLASSES

Multiple inheritances is a process of creating a new class which is derived from more than one base classes. Multiple in heritance hierarchies can be complied, which may lead to a situation in which a derived class inherits multiple times from the same indirect base class.

```
  Class classA
  {
        Protected:
              Int x;
        Public:
              …………
              …………
};
Class classB : public classA
{
        Public:
              ……………
              ……………
};
Class classC : public classA
{
        Public:
              …………..
              …………..
};
Class classD : public classB, public classC
{
        Public:
        …………….     // the data member x comes twice
        …………….
};
```

The data member x is inherited twice in the derived class classD, once through classB and again through classC. By classB and classC into virtual base classes for classD, a copy of the data member x is available only once.

A class may be both an ordinary and a virtual base in the same inheritance structure.

```
  Class classA
  {
        Protected:
              Int x;
        Public:
              …………
              …………
};
Class classB : public virtual classA
{
        Public:
              ……………
              ……………
};
Class classC : public virtual classA
```

```
{
        Public:
                …………..
                …………..
};
Class classD : public classB, public classC
{
        Public:
        …………….     // the data member x comes twice
        …………….
};
```

From the above illustration, it can be inferred that an object of derived class classD will contain tow classA class object, one virtual and one non virtual.

TEMPLATES AND EXCEPTION HANDLING

FUCITON TEMPLATE

Template is a method for writing a single function or class for a family of similar functions or classes in a generic manner. When a single function is written for a family of similar functions called ***function template***. In this function at least one formal argument is generic.

General syntax:

```
         Template<class T> T function_name(T formal arguments)
        {
                ……………..
                …………….
                Return(T);
        }
```

Where the template and class are keywords in C++ and the function template must start with template and the T is a parameterized data type.

Example:

```
Template<class T>
T swap (T &first, T &second)
{
          T temp;
          temp=first;
          first=second;
          second=temp;
          return(0);
}
```

Which supports all the following functions?

```
Char swap(char *, char *);
Int swap(int, int);
Float swap(float, float);
```

Program: A program to demonstrate function template

```
   #include<iostream.h>
   Template<class T> T swap(T &first. T &second)
   {
       T temp;
       temp=first;
       first=second;
       second=temp;
       return(0);
     }
     Int swap(int &a, int &b);
     Void main()
     {
       Int x= 10, y=20;
       Cout<<”Before Swapping”<<endl;
       Cout<<”x value”<<x<<endl;
       Cout<<”y value<<y<<endl;
       Swap(x, y);
       Cout<<”After Swapping”<<endl;
       Cout<<”x value”<<x<<endl;
       Cout<<”y value”<<y<<endl;
     }
```

Output:

```
     Before Swapping
     X value 10y value 20
     After Swapping
     X value 20
     Y value 10
```

CLASS TEMPLATE

In addition to function template, C++ also supports the concept of class templates. By definition, a class template is a class definition that describes a family of related classes,. C++ offers the user the ability to create a class that contains one or more types that are generic or parameterized. The manner of declaring the class template is the same as that of a function template. The keyword template must be inserted as a first word of defining a class template.

General Syntax:

Template <class T>
Class user_ defined_ name
{
    Private:
    …………….
    …………….

```
        }
```

```
        Public:
        …………….
        …………….
};
```

Once the class template has been defined, it is required to instantiate a class object using a specific primitive or user defined type to replace the parameterized types.

A member function of a class template also contains the keyword template whenever it is declared outside the scope of a class definition.

Program: A program to demonstrate class templates.

```
#include<iostream.h>
Template<class T>
Class sample
{
    Private:
            T value, value1, value2;
    Public:
            Void getdata();
            Void sum();
};
Template <class T>
Void sample<T> :: getdata()
{
    Cin>>value1>>value2;
}
Template<class T>
Void sample <T> :: sum()
{
      T value;
      Value=value 1+value2;
      Cout<<"Summation is "<<value<<endl;
}
Void main()
{
      Sample <int> obj1;
      Cout<<"Enter the first object data"<<endl;
      Obj 1.detdata();
      Obj 1.sum();
      Sample<float> obj2;
      Cout<<"Enter the second object data"<<endl;
      Obj2.getdata();
      Obj2.sum();
}
```

Output:

```
      Enter the first object data
```

```
Summation is 30
Enter the second object data
Summation is 3.3
```

EXCEPTION HANDLING

An exception is an error or an unexpected event. The exception handler is a set of codes that executes when an exception occurs. Exception handling is one of the most recently added features and perhaps it may not be supported by much earlier versions of the C++ compilers.

Exception handling in C++ provides a better method by which the caller of a function can be informed that some error condition has occurred. The following keywords are used for handling error functions in C++.

Try
Catch
Through

Wherever a caller of a function finds an error, it is difficult to check or trace these critical errors. In C++, these types of error conditions are handled easily using the above keywords.

Whenever a caller of a function detects an error without exception handling, it is very difficult to handle it in a complex and big software. The program must be developed exception handling in such way that it determines the possible errors the program might encounter and then include codes to hand them.

Exception handling provides another way to transfer control and information from point in the execution of a program to an exception handler. A handler will be invoked only by a throw expression in a code executed in the handler's try block or the function called from the handler 's try block.

General Syntax:

```
Try (expression)
        Catch(exception detector)
{
        ……………
        ……………
}
Throw (expression)
{
        //Error message
        …………..
}
```

The try block is a statement. A throw expression is a unary expression of type void.

DATA FILE OPERATION

FILE

File is a collection of data or a set of characters or may be a text or a program. Basically there are two types of files available in C++: sequential access files and random access files. The sequential files are very easy to create than Random access files. In sequential files the data or text will be stored or read back sequentially. In random access files, data can be accessed and processed randomly.

OPENING AND CLOSING OF FILES

The heard file, fstream.h supports the highly sophisticated input/output stream processing techniques and to implement input/output for the advanced language features such as classes, derived classes, function overloading , virtual function and multiple inheritance.

The following methods are used in C++ to read and write files

Ifstream	to read a stream of object from a specified file
Ofstream	to write a stream of object on a specified file
Fstream	both to read and write a stream of objects on a specified file

The heard file fstream.h is a new class which consists of basic file operation routines and functions. The fstream, Ifstream and Ofstream are called as derived class as these class objects are already defined in the basic input and output class namely <iostream.h>

Example: for open a file (Read mode)

```
#include<fstream.h>
#include<iostream.h>
Void main ()
{
	Ifstream infile;
	Infile.open("data_file");	//Open a file
	………………………
}
```

Example: for write a set of streams in a file (Write mode)

```
#include<fstream.h>
#include<iostream.h>
Void main ()
{
	Ofstream infile;
	Infile.open("data_file");	//Open a file for write mode
	……………………….
}
```

Example: for open a file for Read and Write mode

```
#include<fstream.h>
#include<iostream.h>
Void main ()
{
        Fstream infile;
        Infile.open("data_file", Ios:: in||ios:: out);
        ………………………
}
```

When a file is opened for both reading and writing the I/O streams keep track of two file pointers –one for input operation and other for output operation.

General Syntax:

Void Ifstream:: open(const char* fname, int m=ios::in, int port=filebuf::open port);
Void Ifstream:: open (const char* fname, int m=ios::out, int port=filebuf::open port);
Void Ifstream:: open(const char* fname, int m, int port=filebut:: open port);

The list of member functions used as file attributes for the various kinds of file opening operations:

NAME OF THE MEMBER FUNCTION	**MEANING**
Ios:: in	Open a file for reading
Ios:: out	Open a file for writing
Ios:: add	Append at the end of file
Ios:: atc	Seek to end of a file upon opening Instead of beginning
Ios:: trunc	Delete a file if it exists and recreate it
Ios:: nocreate	Open a file a file does not exist
Ios:: replace	Open a file if a file does exist
Ios binary	Open a file for binary mode; Default is text

For a third argument in Borland C++, refer the following table

VALUE	MEANING
0	Default
1	Read only file
2	Hidden file
4	System file
8	Archieve file

Closing a file

The member function close() is used to close a file which has been opened for file processing such as to read, to write and for both to read and write. The close() member function is called automatically by the destructor functions. However, one may call this member function to close the file explicitly. The close member function will not contain any arguments.

General Syntax:

```
#include<fstream.h>
#include<iostream.h>
Void main ()
{

        Fstream infile;
        Infile.open(“data_ file”,ios:: in||ios:: out);
        …………………….
        Infile.close ();                              //Calling to close the file

}
```

STREAM STATE MEMBER FUNCTIONS

File stream classes inherit a stream state member from the Ios class. The stream state member functions give the information status like end of file has been reached or file open failure and so on. The following stream state member functions are used for checking the open failure if any, when one attempts to open a file from the diskette.

Eof() stream state member function is used to check whether a file pointer has reached the end of a file character or not.

General Syntax:

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{
      Ifstream infile;
      Infile.open(“data_file”);            //Open a file
      While(!infile.enof())
      {
                ……………..
                …………….
      }
      ………………….
      }
```

Fail() stream state member function is used to check whether a file has been opened for input Or output successfully, or any invalid operations are attempted or there is an unrecoverable error. If it fails, it returns a nonzero character.

General Syntax:

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{
        Ifstream infile;
        Infile.open("data_ file");          //Open a file
        While(!infile.enof())
        {
                ………………
                ………………
        }
        …………………
        }
```

Bad() The bad() stream state member function is used to check whether any invalid file operations has been attempted or there is an unrecoverable error. The bad() member function returns a nonzero if it is true; otherwise return a zero.

General Syntax:

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{
        Ifstream infile;
        Infile.open("data_ file");          //Open a file
        If(!infile.enof())
        {
              Cout<<"Open failure"<<endl;
              Exit(1);
        }
        ………………………
}
```

Good() is used to check whether the previous file operation has been successful or not. The good() returns a nonzero if all stream state bits are zero.

General Syntax:

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{
        Ifstream infile;
        Infile.open ("data_ file");          //Open a file
        While (! infile.enof ())
        {
```

```
        ……………………
        ……………………
        }
}
```

READING/WRITING A CHARACTER FROM A FILE

The following member functions are used for reading and writing a character from a specified file.

Get() member function is used to read an alphanumeric character from a specified file.

General Syntax:

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{
        Ifstream infile;
        Infile.open("data_ file");              //Open a file
        While(!infile.enof())
        {
                Ch=infile.get()
                ………………
        }
        ……………………
}
```

Put() member function is used to write a character to a specified file or a specified output stream.

General Syntax:

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{

      Ofstream infile;
      Infile.open("data_ file");              //Open a file in write mode
      While(!outfile.enof())
      {
              Ch=outfile.get();
              Cout.put(ch);                   //Display a character onto a screen.
      }
      …………………..
}
```

Program: A program to write and print a set of character in a file.

```
#include<fstream.h>
#include<iostream.h>
Void main()
{
        //Write some stream in a file
        Ofstream outfile;
        Outfile.open("test");
        Outfile<<"Directorate of Distance and Continuing Education"<<endl;
        Outfile<<"Object oriented design"<<endl;
        Outfile.close();
        //Read a file
        Ifstream infile;
        Infile.open("test");
        Char ch;
        While(!infile.enof())
        Cout<<(ch=infile.get());
}
```

Output:

```
Directorate of Distance and Continuing Education
Object oriented design
```

Program: A program to copy the contents of a text file into another.

```
#include<iostream.h>
#include<fstream.h>
Void main ()
{
        Ofstream outfile;
        Ifstream infile;
        Char ifile[10],ofile [10];
        Cout<<"Enter the input and output filenames"<<endl;
        Cin>.ifile>ofile;
        Infile.open(ifile);
        Outfile.open(ofile);
        If(infile.fail())
        {
        Cout<<"Unable to create a file"<<endl;
        Return;
        }
        Char ch;
        While (! infile. eof ())
        {
                Ch= (char) infile. Get ();
                Outfile.open (ch () ;
        }
        Infile. Close ();
        Outfile.close ();
Cout<<"File Successfully copied"<<endl;
}
```

Output:

```
Test
Testing
File successfully copied
```

BINARY FILE OPERATIONS

A binary file is a sequential access file in which data are stored and read back one after another in the binary format instead of ASCII characters. A binary file contains integer, floating point number, array of structures etc. Binary file processing is well suited for the design and development of a complex database or to read and write a binary information.

The text file created by C++ can be edited by an ordinary editor or by a word processor. The text file can easily be transferred from one computer system to another. On the other hand, a binary file is more accurate for numbers because it stores the exact internal representation of a value. There are no conversions taking place while storing data to file. The binary format data file normally takes less space. However, binary format data file can not be easily transferred from one computer system to another due to variations in the internal representation of the data from one computer to another.

General Syntax:

```
        Infile ("data", ios::binary);
```

Program: A program to open a binary file for storing a set of numbers on a specified file.

```
#include<fstream.h>
#include<iostream.h>
#include<iomanip.h>
Void main ()
{
        Ofstream outfile;
        Char fname [10];
        Float x, y, temp;
        Outfile.open ("bin", Ios :: out||ios:: binary);
        For (int i=0; i<=10; i++)
        Outfile<<I;
        Outfile.close ();
}
```

The bin file contains

```
012345678910
```

STRUCTURES AND FILE OPERATIONS

An array of structures can be stored and accessed using file handling commands. Sometimes, it may be required to store collective structure elements and retrieve them in the similar format.

General Syntax:

```
        Infile<<structure_ var>.data_ member
```

Program: A program to read a data for the structure elements from the keyboard and to store them on a specified file and also displayed in the console.

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
Struct student
{
        Int rollno;
        Int mark;
};
Void main ()
{
        Struct student st [3];
        Int I;
        Fstream infile;
        Char fname [10];
        Cout<<"Enter the file name to store the details"<<endl;
        Cin>>fname;
        Infile.open (fname, ios::out);
        For (i=0; i<3; i++)
        {
        Cout<<"Enter the roll no."<<endl;
        Cin>>st[i].rollno;
        Cout<<"Enter the mark"<<endl;
        Cin>>st[i].mark;
        }
        For (i=0; i<3; i++)
        {
                Infile<<st[i].rollno<<setw96) <<st[i].mark<<endl;
        }
        Infile.close ();
        //Read the data from the file

        Infile.open (fname. Ios:: in);
        Cout<<"Reading from the file"<<endl;
        I=0;
        While (! infile. eof ())
        {
                Infile>>st[i].rollno>>setw (6)>>st[i].mark;
                ++I;
        }
        For (intk=0; k<3; k++)
        {
              Cout<<st[k].rollno<<set (6)<<st[k].mark<<endl;
        }
}
```

Output:

```
Enter the file name to store the details
Jthasvi
Enter the roll no.
10001
Enter the mark
67
Enter the roll no.
10002
Enter the mark
87
Enter the roll no.
10003
Enter the mark
98
Reading from the file
10001       67
10002       87
10003       98
```

CLASS AND FILE OPERATIONS

The header file fstream.h must be included for handling the file input and output operations. The mode of file operations such as to read, to write and both to read and write should be defined. The binary file operations required to handle the input and output are carried out using the member functions get () and put () for insertion and extraction operators.

The member functions read () and write () are used to read and write a stream of objects from a specified file respectively.

General Syntax:

```
        Infile. Read ((char*) &obj, sizeof (obj));
```

Example:

```
        Infile.open ("data", Ios:: in);
        Infile. Read ((char*)&obj, sizeof(obj));
        ………………………………
        Infile.close ();
```

General Syntax:

```
        Infile.write((char*)7obj, sizeof(obj));
```

Example:

```
        Outfile. Open("data", Ios:: out);
        Outfile.write((char*)&obj, sizeof{obj));
        ………………………………
        Outfile.close ();
```

ARRAY OF CLASS OBJECTS AND FILE PERATIONS

An array is a user defined data type whose elements are homogeneous and stored in consecutive memory locations. For practical applications, an array of class objects are essential to construct complex data base systems and hence it is meaningful to study how array of class objects are read and written on a file.

General Syntax:

Infile.write((char*) &obj[i],sizeof(obj[i]));

Program: A program to read a data for the array of class objects from the keyboard and to store them on a specified file and also display in the console.

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
Class student
{
         Public:
         Int rollno;
         Int mark;
};
Void main ()
{
        Student st [3];
        Int I;
        Fstream infile;
        Char fname [10];
        Cout<<"Enter the file name to store the details"<<endl;
        Cin>>fname;
        Infile.open (fname, ios::out);
        For (i=0; i<3; i++)
        {
        Cout<<"Enter the roll no."<<endl;
        Cin>>st[i].rollno;
        Cout<<"Enter the mark"<<endl;
        Cin>>st[i].mark;
        }
        For (i=0; i<3; i++)
        {
                Infile<<st[i].rollno<<setw96) <<st[i].mark<<endl;
        }
        Infile.close ();
        //Read the data from the file
        Infile.open (fname. Ios:: in);
        Cout<<"Reading from the file"<<endl;
        I=0;
        While (! infile. eof ())
        {
              Infile>>st[i].rollno>>setw (6)>>st[i].mark;
              ++I;
        }
        For (intk=0; k<3; k++)
        {
            Cout<<st[k].rollno<<set (6)<<st[k].mark<<endl;
        }
}
```

Output:

```
Enter the file name to store the details
Jthasvi
Enter the roll no.
10001
Enter the mark
67
Enter the roll no.
10002
Enter the mark
87
Enter the roll no.
10003
Enter the mark
98
Reading from the file
10001      67
```

10002	87
10003	98

RANDOM ACCESS FILE PROCESSING

A Sequential access file is very easy to create than a random access file. In the sequential access file, data are stored and retrieved one after another. The file pointer always moves from the starting of the file to the end of the file. On the other hand, a random access file need not necessarily start from the beginning of the file and move towards of end of the file. Random access means moving the file pointer directly to any location in the file instead of moving it sequentially. The random access approach is often used with data base files. In order to perform both reading and modifying an object of a data base, a file should be opened with mode of access for both to read and to write. The header file <fstream.h> is required to declare a random access file. As stated in the previous section that fstream is a class which is based on both the classes of Ifstream and Ofstream. The fstream inherits two file pointers, one for the input buffer and other for the output buffer for handling a random access file both for reading and writing.

The random access file must be opened with the following mode of access.

MODE OF ACCESS	**MEANING**
Ios:: in	In order to read a file
Ios:: out	In order to write a file
Ios:: ate	In order to append
Ios :: binary	Binary format

General Syntax:

Fstream file;
File. open (fname, Ios:: in||ios:;in||ios::out||ios::ate||ios::binary);

It is essential to open a random access file with the following mode of access in order to perform read, write and append. The file should be declared be declared as a binary status as the data members of a class is stored in a binary format.

The fstream inherits the following member functions in order to move the file pointer in and around the data base.

ENUMERATED VALUE	**FILE POSTTION**
Ios:: beg	From the beginning of the file
Ios:: cur	From the current file pointer position
Ios:: end	From the end of the file

Seepage () member function is used to position file operations for random input operations

Example:

```
Infile. Seekg(40);                    //goto byte number 40
Infile. Seekg (40,ios:: beg);            //same as the above
Infile. Seekg (0,ios:: end);             //goto the end of file
Infile. Seekg(0);                        //goto start of the file
Infile. Seekg (-1,ios:: cur);           // file pointers moved back end by one byte.
```

Seekp() member function is used to positive file operations for random output operations.
Tellg () member function is used to check the current position of the input stream.
Tellp () member function is used to check the current position of the output stream.