SRINIVASAN COLLEGE OF ARTS & SCIENCE

(Affiliated Bharathidasan University, Tiruchirappalli)

PERAMBALUR-621212

Department of Computer Science & Information Technology

COURSE MATERIAL

Subject : Programming in C++

Subject Code : 16SCCCS2

Class : I-B.Sc(CS)

Semester : II

CORE COURSE – III

PROGRAMMING IN C++

Objective:

To impart basic knowledge of Programming Skills in C++ language.

Unit I

Principles of object- Oriented Programming – Beginning with C++ - Tokens, Expressions and control structures – Functions is C++

Unit II

Classes and Objects – Constructors and Destructors – New Operator – Operator Overloading and Type Conversions

Unit III

Inheritance : Extending classes – Pointers – Virtual Functions and Polymorphism

Unit IV

Managing console I/O Operations – Working with Files – Templates – Exception Handling

Unit V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

Text Book

1. Balagursamy E, Object Oriented Programming with C++, Tata McGraw Hill Publications, Sixth Edition, 2013

Reference Books

1. Ashok Kamthane, Programming in C++, Pearson Education, 2013.

******

UNIT –I

Introduction:

Programmers write instructions in various programming languages to perform their computation tasks suchas:

(i) Machine levelLanguage
(ii) Assembly levelLanguage
(iii) High levelLanguage

Machine level Language :

Machine code or machine language is a set of instructions executed directly by a computer's central processing unit (CPU). Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory. Every program directly executed by a CPU is made up of a series of such instructions.

Assembly level Language :

An assembly language (or assembler language) is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.

High level Language :

High-level language is any programming language that enables development of a program in much simpler programming context and is generally independent of the computer's hardware architecture. High-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization.

The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, including Ada , Algol, BASIC, COBOL, C, C++, JAVA, FORTRAN, LISP, Pascal, and Prolog. Such languages are considered high-level because they are closer to human languages and farther from machine languages. In contrast, assembly languages are considered low-level because they are very close to machine languages.

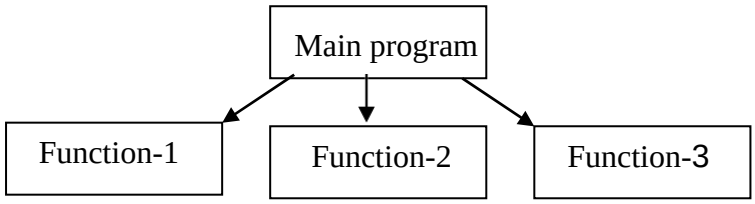
The high-level programming languages are broadly categorized in to two categories:

(iv) Procedure oriented programming(POP)language.
(v) Object oriented programming(OOP)language.

Procedure Oriented Programming Language

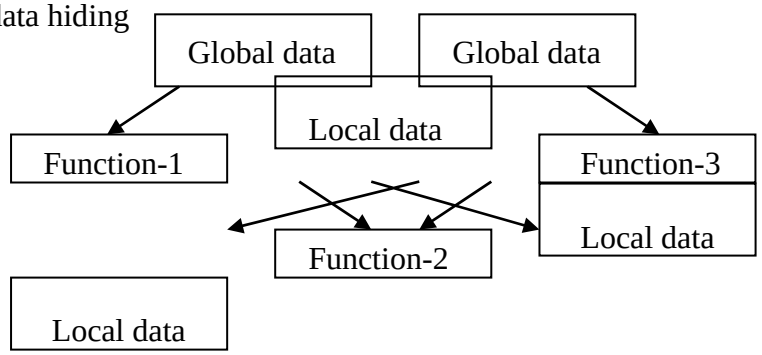
In the procedure oriented approach, the problem is viewed as sequence of things to be done such as reading , calculation and printing.

Procedure oriented programming basically consist of writing a list of instruction or actions for the computer to follow and organizing these instruction into groups known as functions.



The disadvantage of the procedure oriented programming languages is:

1. Global data access
2. It does not model real word problem very well
3. No data hiding

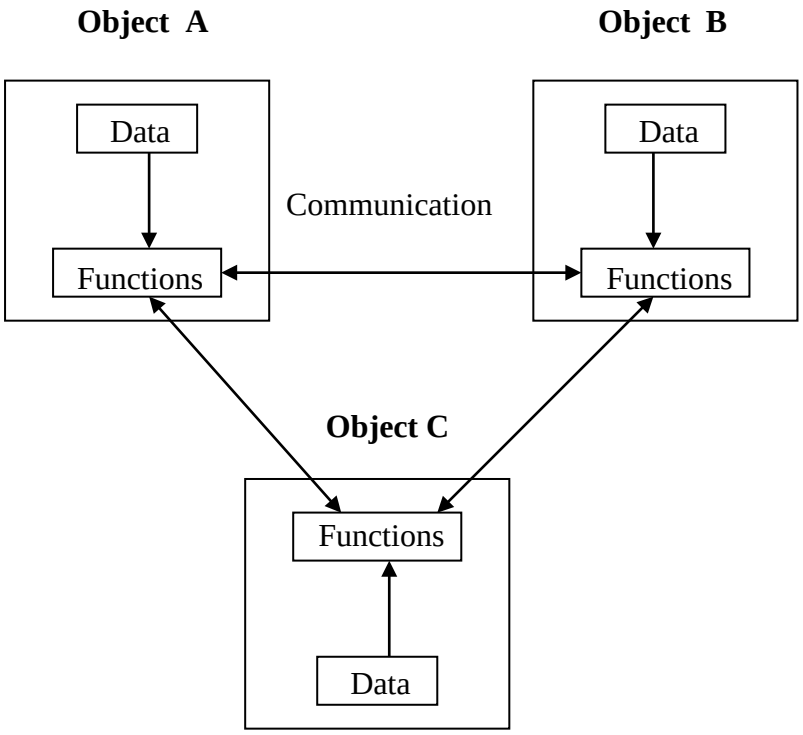


Characteristics of procedure oriented programming:

1. Emphasis is on doing things(algorithm)
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data
4. Data move openly around the system from function to function
5. Function transforms data from one form to another.
6. Employs top-down approach in program design

Object Oriented Programing

"Object oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand".



Features of the Object Oriented programming

1. Emphasis is on doing rather than procedure.
2. programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and can't be accessed by external functions.
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added.
8. Follows bottom-up approach in program design.

BASIC CONCEPTS OF OBJECTS ORIENTED PROGRAMMING

1. Objects
2. Classes
3. Data abstraction andencapsulation
4. Inheritance
5. Polymorphism
6. Dynamicbinding
7. Message passing

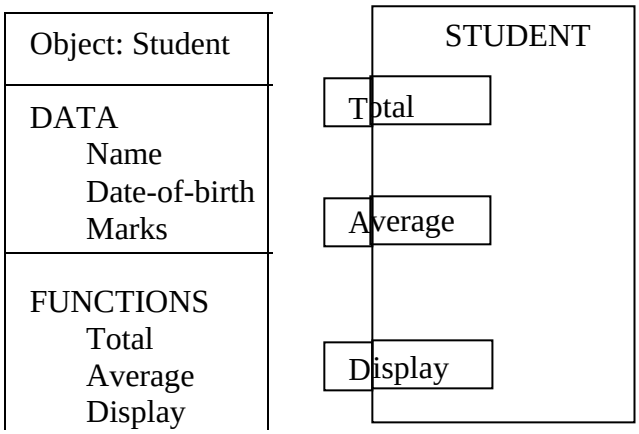
OBJECTS

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle.

The fundamental idea behind object oriented approach is to combine both data and function into a single unit and these units are called objects.

The term objects means a combination of data and program that represent some real word entity. For example: consider an example named Amit; Amit is 25 years old and his salary is 2500. The Amit may be represented in a computer program as an object. The data part of the object would be (name: Amit, age: 25, salary:2500)

The program part of the object may be collection of programs (retrive of data, change age, change of salary). In general even any user –defined type-such as employee may be used. In the Amit object the name, age and salary are called attributes of theobject.



CLASS:

A group of objects that share common properties for data part and some program part are collectively called as class.

In C ++ a class is a new data type that contains member variables and member functions that operate on the variables.

DATA ABSTRACTION :

Abstraction refers to the act of representing essential features without including the back ground details or explanations. Classes use the concept of abstraction and are defined as size, width and cost and functions to operate on the attributes.

DATA ENCAPSALATION :

The wrapping up of data and function into a single unit (called class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the objects data and the program.

INHERITENCE :

Inheritance is the process by which objects of one class acquire the properties of another class. In the concept of inheritance provides the idea of reusablity. This mean that we can add additional features to an existing class with out modifying it. This is possible by desining a new class will have the combined features of both theclasses.

POLYMORPHISIM:

Polymorphism means the ability to take more than one form. An operation may exhibit different instance. The behaviour depends upon the type of data used in the operation.

A language feature that allows a function or operator to be given more than one definition. The types of the arguments with which the function or operator is called determines which definition will be used.

Overloading may be operator overloading or function overloading.

It is able to express the operation of addition by a single operater say '+'. When this is possible you use the expression x + y to denote the sum of x and y, for many different types of x and y; integers , float and complex no. You can even define the + operation for two strings to mean the concatenation of the strings.

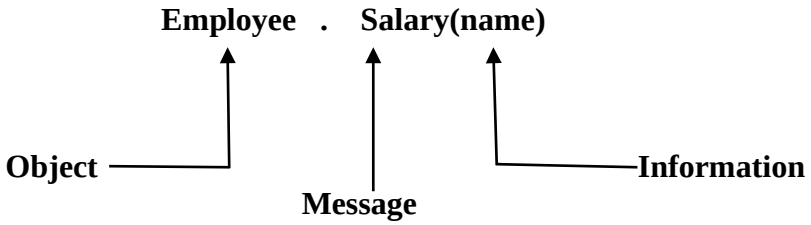
DYNAMIC BINDING :

Binding refers to the linking of a procedure call to the code to the executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known untill the time of the call at run-time. It is associated with a polymorphic reference depends upon the dynamic type of thatreference.

MESSAGE PASSING :

An object oriented program consists of a set of objects that communicate with each other.

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 the object, the name of the function (message) and information to be sent.



BENEFITS OF OOP:

Oop offers several benefits to both the program designer and the user. Object-oriented contributes to the solution of many problems associated with the development and quality of software products. 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 higherproductivity.
3. This principle of data hiding helps the programmer to build secure programs that can't be invaded by code in other parts of theprogram.
4. It is possible to have multiple instances of an object to co-exist with out anyinterference.
5. It is easy to partition the work in a project based onobjects.
6. Object-oriented systems can be easily upgraded from small to largesystems.
7. Message passing techniques for communication between objects makes the interface description with external systems muchsimpler.
8. Software complexity can be easilymanaged.

APPLICATION OF OOP:

The most popular application of oops up to now, has been in the area of user interface design such as windows. There are hundreds of windowing systems developed using oop techniques.

Real business systems are often much more complex and contain many more objects with complicated attributes and methods. Oop is useful in this type of applications because it can simplify a complex problem. The promising areas for application of oop includes.

1. Real – Timesystems.
2. Simulation andmodeling
3. Object orienteddatabases.
4. Hypertext,hypermedia andexpertext.

5. Al and expertsystems.
6. Neural networks and parallelprogramming.
7. Dicision support and office automationsystems.
8. CIM / CAM / CADsystem.

Basics of C++

C ++ is an object oriented programming language, C ++ was developed by Jarney Stroustrup at AT & T Bell lab, USA in early eighties. C ++ was developed from c and simula 67 language. C ++ was early called 'C with classes'.

C++ Comments:

C++ introduces a new comment symbol //(double slash). Comments start with a double slash symbol and terminate at the end of line. A comment may start any where in the line and what ever follows till the end of line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multi line comments can be written as follows:

```
// this is an example of
// c++ program
// thank you
```

The c comment symbols /* ….*/ are still valid and more suitable for multi line comments.

```
/* this is an example of c++ program */
```

Output Operator:

The statement cout <<"Hello, world" displayed the string with in quotes on the screen. The identifier cout can be used to display individual characters, strings and even numbers. It is a predefined object that corresponds to the standard output stream. Stream just refers to a flow of data and the standard Output stream normally flows to the screen display. The cout object, whose properties are defined in iostream.h represents that stream. The insertion operator << also called the 'put to' operator directs the information on its right to the object on its left.

Return Statement:

In C++ main ( ) returns an integer type value to the operating system. Therefore every main ( ) in C++ should end with a return (0) statement, otherwise a warning or an error might occur.

Input Operator:

The statement

```
cin>> number 1;
```

is an input statement and causes. The program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** is a predefined object in C++ that corresponds to the standard input stream. Here this stream represents the key board.

The operator >>is known as get from operator. It extracts value from the keyboard and assigns it to the variable on itsright.

Cascading Of I/O Operator:

```
cout<<”sum=”<<sum<<”\n”;
cout<<”sum=”<<sum<<”\n”<<”average=”<<average<<”\n”;
cin>>number1>>number2;
```

Structure Of A Program :

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout <<"Hello World!";
  return 0;
}
```

Output:-Hello World!

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have justwritten:

// my first program in C++

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

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive #include<iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones tobe

executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.
The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.
Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it isexecuted.
cout <<"Hello World!";
This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.
cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is thescreen).
cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.
Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).
return 0;
The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.
You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the mainfunction.
The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, insteadof

```
int main ()
{
  cout <<" Hello World!";
  return 0;
}
```

We could have written:

```
int main ()
{
cout <<"Hello World!";
return 0;
}
```

All in just one line and this would have had exactly the same meaning as the previous code.
In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The divisionof

code in different lines serves only to make it more legible and schematic for the humans that may readit.

Let us add an additional instruction to our first program:

```
// my second program in C++
#include <iostream>
using namespace std;

int main ()
{
  cout <<"Hello World! ";
  cout <<"I'm a C++ program";
  return 0;
}
```

Output:-Hello World! I'm a C++ program

In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since main could have been perfectly valid defined this way:

```
int main ()
{
 cout <<" Hello World! ";
cout <<" I'm a C++ program ";
 return 0;
}
```

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
  cout <<"Hello  World!"; cout
  <<"I'm a C++program";
  return0;
}
```

And the result would again have been exactly the same as in the previous examples.
Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (;).

STRUCTURE OF C++ PROGRAM

- **Include files**
- **Classdeclaration**
- **Class functions,definition**
- **Main functionprogram**

Example :-

```
# include<iostream.h>

class person
```

```
{
char name[30];
int age;
public:
        void getdata(void);
        void display(void);
};


void person :: getdata ( void )
{
        cout<<"enter name";
        cin>>name;
        cout<<"enter age";
        cin>>age;
}

void display()
{
        cout<<"\n name:"<<name;
        cout<<"\n age:"<<age;
}

int main( )
{

person p;
p.getdata();
p.d isplay();
return(0);
```

TOKENS:

The smallest individual units in program are known as **tokens.** C++ has the following tokens.

i. Keywords
ii. Identifiers
iii. Constants
iv. Strings
v. Operators

KEYWORDS:

The keywords implement specific C++ language feature. They are explicitly reserved identifiers and can't be used as names for the program variables or other user defined program elements. The keywords not found in ANSI C are shown in red letter.

C++ KEYWORDS:

Asm	double	new	switch
Auto	else	operator	template
Break	enum	private	this
Case	extern	protected	throw
Catch	float	public	try
Char	for	register	typedef
Class	friend	return	union
Const	goto	short	unsigned
Continue	if	signed	virtual
Default	inline	sizeof	void
Delete	long	struet	while

IDENTIFIERS:

Identifiers refers to the name of variable , functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers.

The following rules are common for both C and C++.

1. Only alphabetic chars, digits and under score arepermitted.
2. The name can't start with adigit.
3. Upper case and lower case letters aredistinct.
4. A declared keyword can't be used as a variablename.

In ANSI C the maximum length of a variable is 32 chars but in c++ there is no bar.

BASIC DATA TYPES IN C++

1. **User Defiened Datatype**
   - **-Structure**
   - **-Union**
   - **-Class**
   - **-Enumeration**

2. **Derived Datatype**
   - **-Array**
   - **-Function**
   - **-Pointer**
3. **Built-in-Datatypes**
   - **Integral-int,char**
   - **Floating Point-float,double**
   - **Void**

Both C and C++ compilers support all the built in types. With the exception of void the basic datatypes may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long and short may applied to character and integer basic data types. However the modifier long may also be applied to double.

Data types in C++ can be classified under various categories.

TYPE	BYTES	RANGE
char	1	-128 to – 127
usigned	1	0 to 265
sgned char	1	-128 to 127
Int	2	-32768 to 32768
unsigned int	2	0 to 65535
singed int	2	-32768 to 32768
short int	2	-32768 to 32768
long int	4	-2147483648 to 2147483648
signed long int	4	-2147483648 to 2147483648
unsigned long int	4	0 to 4294967295

float	4	3.4E-38 to 3.4E+38
double	8	1.7E -308 to 1.7E +308
long double	10	3.4E-4932 to 1.1E+ 4932

The type void normally used for:

1) To specify the return type of function when it is not returning anyvalue.
2) To indicate an empty argument list to afunction.

Example:

Void function(void);

Another interesting use of void is in the declaration of genetic pointer

Example:

Void *gp;

Assigning any pointer type to a void pointer without using a cast is allowed in both C and ANSI C. In ANSI C we can also assign a void pointer to a non-void pointer without using a cast to non void pointer type. This is not allowed in C++.

Example:

v**oid *ptr1;**

void *ptr2;

Are valid statement in ANSI C but not in C++. We need to use a cast operator.

p**tr2=(char *) ptr1;**

USER DEFINED DATA TYPES:

STRUCTERS AND CLASSES

We have used user defined data types such as struct,and union in C. While these more features have been added to make them suitable for object oriented programming. C++ also permits us to define another user defined data type known as class which can be used just like any other basic data type to declare a variable. The class variables are known as objects, which are the central focus of oops.

ENUMERATED DATA TYPE:

An enumerated data type is another user defined type which provides a way for attaching names to number, these by increasing comprehensibility of the code. The enum keyword

automatically enumerates a list of words by assigning them values 0,1,2 and soon. This facility provides an alternative means for creating symbolic.

Example:

enum shape {circle,square,triangle}

enum colour{red,blue,green,yellow}

enum position{off,on}

The enumerated data types differ slightly in C++ when compared with ANSI C. In C++, the tag names shape, colour, and position become new type names. That means we can declare new variables using the tag names.

Example:

Shape ellipse;//ellipse is of type shape

colour background ; // back ground is of type colour

ANSI C defines the types of enums to be ints. In C++,each enumerated data type retains its own separate type. This means that C++ does not allow an int value to be automatically converted to an enum.

Example:

colour background =blue; //vaid

colour background =7; //error in c++

colour background =(colour) 7;//ok

How ever an enumerated value can be used in place of an int value.

Example:

int c=red ;//valid, colour type promoted to int

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second and so on. We can also write

enum color {red, blue=4,green=8};

enum color {red=5,blue,green};

C++ also permits the creation of anonymous enums (i.e, enums without tag names)

Example:

enum{off,on};

Here off is 0 and on is 1.these constants may be referenced in the same manner as regular constants.

Example:

int switch-1=off;

int switch-2=on;

ANSI C permits an enum defined with in a structure or a class, but the enum is globally visible. In C++ an enum defined with in a class is local to that class.

SYMBOLIC CONSTANT:

There are two ways of creating symbolic constants in c++.

1. using the qualifierconst.
2. defining a set of integer constants using enumkeywords.

In both C and C++, any value declared as const can't be modified by the program in any way. In C++, we can use const in a constant expression. Such as

```
const int size = 10 ;
char name (size) ;
```

This would be illegal in C. const allows us to create typed constants instead of having to use #defme to create constants that have no type information.

const size=10;

Means

const int size =10;

C++ requires a const to be initialized. ANSI C does not require an initializer, if none is given, it initializes the const to 0.

InC++constvaluesare localandinANSICconstvaluesareglobal.Howevertheycanbemadelocal made localbydeclaringthemasstatic.InC++ifwewant tomakeconstvalueasglobalthendeclareasextern storageclass.

Ex: externalconsttotal=100;

Anothermethod ofnamingintegerconstantsisasfollows:-

enum {x,y,z};

DECLARATION OF VARIABLES:

In ANSIC C all the variable which is to be used in programs must be declared at the beginning of the program.ButinC++wecandeclarethevariablesanywhoseintheprogramwhere itrequires.This makesthe program much easier to write and reduces the errors that may be caused by having to scan back and forth. It alsomakestheprogrameasiertounderstand becausethevariablesaredeclaredinthecontextoftheiruse.

Example:

```
main()
 {
        float x,average;
        floatsum=0;
        for(int i=1;i<5;i++)
              {
               cin>>x;
               sum=sum+x
              }
               float average;
              average=sum/x;
              cout<<average;
 }
```

REFERENCE VARIABLES:

C++interfaces a new kind of variable known as the reference variable. A references variable provides an alias.(alternative name) for a previously defined variable. For example ,if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent the variuble.

A reference variable is created as follows:

Synatx: Datatype & reference –name=variable name;

Example:

```
float total=1500;
float &sum=total;
```

Here sum is the alternative name for variables total, both the variables refer to the same data object in the memory .

A reference variable must be initialized at the time of declaration .

Note that C++ assigns additional meaning to the symbol &here &is not an address operator .The notation float &means reference to float.

Example:

```
int n[10];
int &x=n[10];
char &a='\n';
```

OPERATORS IN C++ :

C++ has a rich set of operators. All C operators are valid in C++ also. In addition. C++ introduces some new operators.

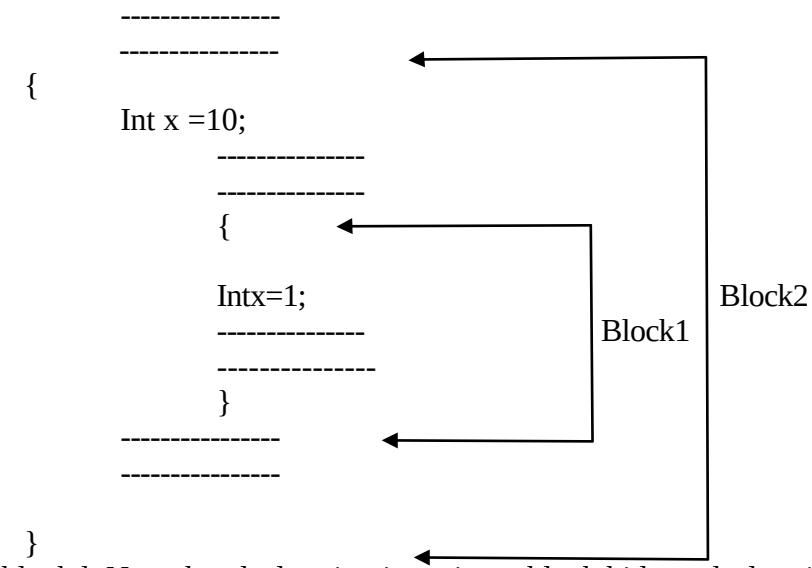
<<	insertionoperator
>>	extractionoperator
::	scope resolutionoperator
::*	pointer to memberdeclarator
*	pointer to memberoperator
.*	pointer to memberoperator
Delete	memory releaseoperator
Endl	line feedoperator
New	memory allocationoperator
Setw	field widthoperator

SCOPE RESOLUTION OPERATOR:

LikeC,C++ isalsoablock-structuredlanguage.Block-structuredlanguage.Blocksand scopes can be used in constructing programs. We know same variables can be declared in different blocks because the variables declared in blocks are local to thatfunction.

Blocks in C++ are oftennested.

Example:



Block2 contained in block l .Note that declaration in an inner block hides a declaration of the same variable in an outer block and therefore each declaration of x causes it to refer to a different data object . With in the inner block the variable x will refer to the data object declared there in.

In C,the global version of a variable can't be accessed from with in the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator .This can be used to uncover a hidden variable.

Syntax: **: : variable–name;**

Example:

```
#include <iostrcam.h>
        int m=10;
        main()
        {
        int m=20;
        {
        int k=m;
        int m=30;
        cout<<”weareininnerblock”;
        cout<<"k="<<k<<endl;
        cout<<"m="<<m<<endl;
        cout<<":: m="<<::m<<endl;
        }
        cout<<”\nweareinouterblock\n”;
        cout<<"m="<<m<<endl;
        cout<<":: m="<<::m<<endl;
         }
```

Memory Management Operator

C uses malloc and calloc functions to allocate memory dynamically at run time . Similarly it uses the functions Free()to freedynamicallyallocatedmemory. Weusedynamic allocationtechniqueswhenitisnotknownin advancehowmuchofmemoryspaceasneeded.

C++ also support those functions it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

Thenewoperatorcanbeusedtocreateobjectsofanytype.**Syntax:** **pointer-**

variable =newdatatype;

Example:

p=new int; q=new int;

Where p is a pointer of type int and q is a pointer of type float.

int *p=new int;

float *p=newfloat;

Subsequently, the statements

*p=25;

*q=7.5;

Assign25tothenewlycreatedint objectand7.5tothefloatobject.Wecanalsoinitializethememory using the newoperator.

Syntax:

int *p=ne\v int(25);
float *q =new float(7.5);

newcanbeusedtocreateamemoryspaceforanydatatype includinguserdefinedsuchas arrays,structures,andclasses.Thegeneralformforaone-dimensionalarrayis:

pointer-variable =new data types [size];

creates a memory space for an array of 10 integers.

If a data object is no longer needed, it is destroyed to release the memory space for reuse.

Syntax: delete pointer-variable;

Example:

delete p;
delete q;

If we want to free a dynamically allocated array ,we must use the following form ofdelete.

delete [size] pointer-variable;

or

delete [ ] pointer variable;

MANIPULATERS:

Manipulators are operator that are used to format the data display. The most commonly manipulators are endl and setw.

The endl manipulator, when used in an output statement, causes a line feed to be insert.(just like \n)

Example:

```
cout<<"m="<<m<<endl;
cout<<"n="<<n<<endl;
cout<<"p="<<p<<endl;
```

Ifweassumethevaluesofthevariablesas2597,14and175respectively

m=2597; n=14;
p=175

Itwaswanttoprint allnosinright justifiedwayusesetwwhichspecifyacommonfieldwidth for all thenos.

Example:

```
cout<<setw(5)<<sum<<endl;
cout<<setw(10)<<"basic"<<setw(10<<basic<<endl;
Cout<<setw(10)<<"allowance"<<setw(10<<allowance<<endl;
cout<<setw(10)<<"total="<<setw(10)<<total;
```

CONTROL STRUCTURES:

Like c,c++, supports all the basic control structures and implements them various control statements.

The if statement:

The if statement is impklemented in two forms:

1. simple ifstatement
2. if… elsestatement

Simple if statement:

```
if (condition)
{
Action;
}
```

If.. else statement

```
If (condition)
Statment1
Else
Statement2
```

The switch statement

Thisisamultiple-branchingstatementwhere,basedonacondition,thecontrolistransferredtooneofthemany possiblepoints;

Switch(expr)

{

case 1:

action1;

break;

case 2:

action2;

break;

..

..

default:

message

}

The whilestatement:

Syn:

While(condition)

{

Stements

}

The do-while statement:

Syn:

do

{

Stements

} while(condition);

The for loop:

for(expression1;expression2;expression3)

{

Statements;

Statements;

}

FUNCTION IN C++ :

The main( ) Functon ;

ANSI does not specify any return type for the main ( ) function which is the starting point for the execution of a program . The definition of main( ) is :-

```
main()
{
//main program statements
 }
```

Thisispropertyvalidbecausethemain()inANSICdoesnotreturnanyvalue.InC++,themain()returnsavalueof typeinttotheoperatingsystem.Thefunctionsthathaveareturnvalueshouldusethereturnstatementforterminating. Themain()functioninC++isthereforedefinedasfollows.

```
int main( )
{
--------------
--------------
return(0)
}
```

Since the return type of functions is int by default, the key word int in the main( ) header is optional.

INLINE FUNCTION:

To eliminate the cost of calls to small functions C++ proposes a new feature called inline function. An inline function is a function that is expanded inline when it is invoked .That is the compiler replaces the function call with the corresponding functioncode.

The inline functions are defined as follows:-

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

Example:

```
inline double cube (doublea)
{
return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c=cube(3.0);
d=cube(2.5+1.5);
```

remember that the inline keyword merely sends a request, not a command to the compliler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values if a loop, a switch or a go toexists.

2. for function s not returning values, if a return statementexists.
3. if functions contain staticvariables.
4. if inline functions arerecursive,.

Example:

```
#include<iostream.h>
#include<stdio.h>
inline float mul(float x, float y)
{
        return(x*y);
}
inline double div(double p.double q)
{
         return(p/q);
}



main( )
{
        float a=12.345;
        float b=9.82;
        cout<<mul(a,b)<<endl;
        cout<<div (a,b)<<endl;
}
```

output:-

```
121.227898
  1.257128
```

DEFAULT ARGUMENT:-

C++ allows us to call a function with out specifying all its arguments.In such cases, the function assigns a default value to the parameter which does not have a matching aguments in the function call.Default values are specified when the function is declared .The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Example: float amount (float principle, int period ,floatrate=0.15);

The default value is specified in a manner syntactically similar to a variable initialization .The above prototype declares a default value of 0.15 to the argument rate. A subsequent function calllike

value=amount(5000,7); //one argument missing

passes the value of 5000 to principle and 7 to period and then lets the function, use default value of 0.15 forrate.

The call:- value=amount(5000,5,0.12);

//no missing argument passes an explicite value of 0.12 rate.

One important point to note is that only the trailing arguments can have default values. That is, we must add default from right to left .We cannot provide a default to a particular argument in the middle of an argumentlist.

```
Example:-   int mul(int i, int j=5,int k=10);//illegal
            int mul(int i=0,int j,int k=10);//illegal
            int mul(int i=5,intj);//illegal
            int mul(int i=2,int j=5,int k=10);//illegal
```

Default arguments are useful in situation whose some arguments always have the some value. For example,bank interest may retain the same for all customers for a particular period of deposit.

Example:

```
#include<iostream.h>
#include<stdio.h>
mainQ
{
float amount;
float value(float p,int n,float r=0.15);
void printline(char ch='*',int len=40);
printline( );
amount=value(5000.00,5);
cout<<"\n final value="<<amount<<endl;
printline('=');
//function definitions
float value (float p,int n, float r)
{
        float si;
        si=(p*n*r)/100;
        return(si);
}
void printline (char ch,int len)
        {
        for(inti=l;i<=len;i++)
        cout<<ch<<endl;
}
```

output:-

```
* * * * * * * * * * * * * * * *
 final value=10056.71613
= = = = = = = = = = = = = = = =
```

Advantage of providing the default arguments are:

1. We can use default arguments to add new parameters to the existingfunctions.
2. Default argument s can be used to combine similar functions intoone.

CONST ARGUMENT:-

In C++, an argument to a function can be declared as unit as const as shown below.

```
int strlen(const char *p);
int length(const string&s);
```

The qualifier const tells the compiler that the function should not modify the argument .the compiler will generate an error when this condition is violated .This type of declaration is significant only when we pass arguments by reference or pointers.

FUNCTION OVERLOADING:

Overloading refers to the use of the same thing for different purposes . C++ also permits overloading functions .This means that we can use the same function name to creates functions that perform a variety of different tasks. This is known as function polymorphism in oops.

Using the concepts of function overloading , a family of functions with one function name but with different argument lists in the functions call .The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

For example an overloaded add() function handles different types of data as shown below.

```
//Declaration
        int add(int a, int b); //prototype 1
        int add (int a, int b, int c); //prototype 2
        double add(double x, double y); //prototype 3
        double add(double p , double q); //prototype4


//function call
cout<<add(5,10); //uses prototype 1
cout<<add(15,10.0); //uses prototype 4
cout<<add(12.5,7.5); //uses prototype 3
cout<<add(5,10,15); //uses prototype 2
cout<<add(0.75,5); //uses prototype 5
```

A function call first matches the prototype having the same no and type of arguments and then calls the appropriate function for execution.

The function selection invokes the following steps:-

a) The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.
b) If an exact match is not found the compiler uses the integral promotions to the actual arguments such as:
   char to int
   float to double
   to find amatch

c)When either of them tails ,the compiler tries to use the built in conversions to the actual arguments and them uses the function whose match is unique . If the conversion is possible to have multiple matches, then the compiler will give errormessage.

Example:

```
long square (long n);
double square(double x);
```

A function call suchas:- square(lO)

Will cause an error because int argument can be converted to either long or double .There by creating an ambiguous situation as to which version of square( )should be used.

PROGRAM

```
#include<iostream.h>
int volume(double,int);
double volume( double , int );
double volume(longint ,int ,int);
        main( )
        {
                cout<<volume(10)<<endl;
                cout<<volume(10)<<endl; cout<<volume(10)<<endl;
        }
int volume( ini s)
    {
    return (s*s*s); //cube
     }
double volume( double r, int h)
        {
        return(3.1416*r*r*h); //cylinder
        }
long volume (longint 1, int b, int h)
        {
        return(1*b*h); //cylinder
        }

                output:- 1000
                         157.2595
                         112500
```

UNIT-II

CLASS:-

Class is a group of objects that share common properties and relationships .In C++, a class is a new data type that contains member variables and member functions that operates on the variables. A class is defined with the keyword class. It allows the data to be hidden, if necessary from external use. When we defining a class, we are creating a new abstract data type that can be treated like any other built in datatype.

Generally a class specification has two parts:-

a) Class declaration
b) Class functiondefinition

the class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

Syntax:-

```
class class-name
    {
    private:
        variable declarations;
        function declaration ;
    public:
        variable declarations;
        function declaration;
        };
```

The members that have been declared as private can be accessed only from with in the class. On the other hand , public members can be accessed from outside the class also. The data hiding is the key feature of oops. The use of keywords private is optional by default, the members of a class are private.

The variables declared inside the class are known as data members and the functions are known as members mid the functions. Only the member functions can have access to the private data members and private functions. However, the public members can be accessed from the outside the class. The binding of data and functions together into a single class type variable is referred to as encapsulation.

Syntax:-

```
class item
{
    int member;
    float cost;
public:
    void getldata (int a ,float b);
    void putdata (void);
```

The class item contains two data members and two function members, the data members are private by default while both the functions are public by declaration. The function getdata() can be used to assign values to the member variables member and cost, and putdata() for displaying their values . These functions provide the only access to the data members from outside the class.

CREATING OBJECTS:

Once a class has been declared we can create variables of that type by using the classname.

Example:

item x;

creates a variables x of type item. In C++, the class variables are known as objects. Therefore x is called an object of type item.

item x, y ,z also possible.

```
class item
{
-----------
-----------
-----------
}x ,y ,z;
```

would create the objects x ,y ,z of type item.

ACCESSING CLASS MEMBER:

The private data of a class can be accessed only through the member functions of that class. The main() cannot contains statements that the access number and cost directly.

Syntax:

object name.function-name(actualarguments);

Example:- x.getdata(100,75.5);

It assigns value 100 to number, and 75.5 to cost of the object x by implementing the getdata() function .

similarly the statement

x. putdata ( ); //would display the values of data members.

x. number = 100 is illegal .Although x is an object of the type item to which number belongs , the number can be accessed only through a member function and not by the objectdirectly.

Example:

```
class xyz
{
        Intx;
        Inty;
public:
        intz;
};
---------
----------
xyz p;
p.x=0;          error . x isprivate
p,z=10;         ok ,z ispublic
```

DEFINING MEMBER FUNCTION:

Member can be defined in two places

- Outside the class definition
- Inside the classfunction

OUTSIDE THE CLASS DEFlNAT1ON;

Member function that are declared inside a class have to be defined separately outside the class.Their definition are very much like the normal functions.

An important difference between a member function and a normal function is that a member function incorporates a membership.Identify label in the header. The 'label' tells the compiler which class the function belongsto.

Syntax:

```
return type class-name::function-name(argument declaration )
        {
        function-body
        }
```

The member ship label class-name :: tells the compiler that the function function - name belongs to the class class-name . That is the scope of the function is restricted to the class-name specified in the header line. The :: symbol is called scope resolutionoperator.

Example:

```
void item :: getdata (int a , float b )
        {
        number=a;
        cost=b;
        }
void item :: putdata ( void)
        {
        cout<<"number=:"<<number<<endl;
        cout<<"cost="<<cost<<endl;
        }
```

The member function have some special characteristics that are often used in the program development.

- Several different classes can use the same functionname.The "membership label"willresolvetheirscope,memberfunctionscanaccesstheprivatedataoftheclass .A non member function can't do so.
- A member function can call another member function directly, without using the dot operator.

INSIDE THE CLASS DEF1NATION;

Another method of defining a member function is to replace thefunction declaration by the actual function definition inside the class.

Example:

```
class item
{
        Intnumber;
        float cost;
public:
        void getdata (int a ,float b);
        void putdata(void)
        {
                cout<<number<<endl; cout<<cost<<endl;
        }
        };
```

A C++ PROGRAM WITHCLASS:

```
# include< iostream. h>
  class item
        {
                int number;
                float cost;
        public:
                void getdata ( int a , float b);
                void putdala ( void)
                        {
                        cout<<“number:”<<number<<endl;
                        cout<<”cost :”<<cost<<endl;
                        }
                        };
  void item :: getdata (int a , float b)
                        {
                        number=a;
                        cost=b;
                        }
                main ( )
                {
                        item x;
                        cout<<”\nobjectx”<<endl;
                        x. getdata( 100,299.95);
                        x .putdata();
                        item y;
                        cout<<”\n object y”<<endl;
                        y. getdata(200,175.5);
                        y. putdata();
                        }
```

Output: **object x**
number100

cost=299.950012
object -4
cost=175.5

Q.

Write a simple program using class in C++ to input subject mark and printsit.

ans:

```
class marks
{
        private :
                int ml,m2;
                public:
                void getdata();
                void displaydata();
};
void marks: :getdata()
{
        cout<<”enter 1st subject mark:”;
        cin>>ml;
        cout<<”enter 2nd subject mark:”;
        cin>>m2;
}
void marks: :displaydata()
{
        cout<<”Ist subject mark:”<<ml<<endl ;
        cout<<”2nd subject mark:”<<m2;
}
voidmain()
{
clrscr();
marks x;
x.getdata();
x.displaydata();

}
```

NESTING OF MEMBER FUNCTION;

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
#include <iostream.h>
class set
{
        int m,n;
        public:
                void input(void);
                void display (void);
                void largest(void);
};
int set::largest (void)
{
        if(m>n)
                return m;
        else
                return n;
}
voidset::input(void)
{
        cout<<"input values of m and n:";
        cin>>m>>n;
}
void set::display(void)
{
cout<<"largestvalue="<<largest()<<"\n";
}
void main()
{
        set A;
        A.input( );
        A.display( );
}
```

output:

Input values of m and n:

3017

largest value= 30

Private member functions:

Although it is a normal practice to place all the data items in a private section and all the functions in public, some situations may require contain functions to be hidden from the outside calls. Tasks such as deleting an account in a customer file or providing increment to and employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the privatesection.

A private member function can only be called by another function that is a member of its class. Even an object can not invoke a private function using the dot operator.

```
Classsample
{
        intm;
        void read (void);
        void write (void);
};
```

if si is an object of sample, then

```
s.read();
```

is illegal. How ever the function read() can be called by the function update ( ) to update the value of m.

```
void sample :: update(void)
        {
        read( );
        }
```

```
#include<iostream.h>

class part
{
private:
intmodelnum,partnum;
float cost;
public:
void setpart ( int mn, int pn ,float c)
{
modelmim=mn;
partnum=pn;
cost=e;
}
void showpart ( )
{
Cout<<endl<<"model:"<<modelnum<<end1;
Cout<<"num:"<< partnum <<endl
Cout<<"cost:"<<"$"<cost;
}
};
void main()
{
part pl,p2;
p1.setpart(644,73,217.55);
p2.setpart(567,89,789.55);
pl.showpart();
pl.showpart();
}
```

output:- model:644

num:73

cost: $217550003

model: 567

num:89

cost: $759.549988

```
#indude<iostream.h>
        classdistance
        {
        private:
                int feet;
                floatinches;
        public:
                void setdist ( int ft, float in)
                        {
                                feet=ft;
                                inches=in;
                        }
                void    getdist()
                {
                cout<<”enter feet:”;
                cin>>feet;
                cout<<”enter inches:”;
                cin>>inches;
                }
                void showdist()
                {
                cout<< feet<<”_”inches«endl;
                }
                };
                void main( )
                {
                distance dl,d2;
                d1.setdist(1 1,6.25);
                d2.getdata();
                cout<<endl<<”dist:”<<d 1 .showdist();
                cout<<”\n”<<”dist2:”;
                d2.showdist();
                }
```

output:- enter feet: 12

enter inches:6.25

dist 1:”11’-6.1.5”

dist 2: 12’-6.25”

ARRAY WITH CLASSES:

```
#include<iostream.h>
#include<conio.h>
class employee
{
private:
        char name[20];
        int age,sal;
public:
        void getdata();
        void putdata();
};
void employee : : getdata ()
    {
        cout<<"enter name :";
        cin>>name;
        cout<<"enter age :";
        cin>>age;
        cout<<"enter salary:";
        cin>>sal;
        return(0);
    }
    void employee : : putdata ( )
    {
    cout<<name <<endl;
    cout<<age<<endl;
    cout<<sal<<endl;
    return(0);
    }
    intmain()
    {
```

```
employee emp[5]:
for( int i=0;i<5;i++)
{
emp[i].getdata();
}
cout<<endl;
for(i=0;i<5;i++)
{
emp[i].putdata();
}
getch();
return(0);
}
```

ARRAY OF OBJECTS:-

```
#include<iostream.h>
#include<conio.h>
        class emp
                {
                private:
                        char name[20];
                        int age,sal;
                public:
                        void    getdata( );
                        void    putdata( );
                                };
void    emp : : getdata()
                {
                coul<<"enter empname": .
                cin>>name;
                cout<<"enter age:"<<endl;
                cin>>age;
                cout<<"enter salun :";
```

```
                cin>>sal;
            }
void    emp :: putdata()
        {
        cout<<"emp name:"<<name<<endl;
        cout<<"emp age:"<<age<<endl;
        cout<<"emp salary:"<<sal;
        }


        void    main()
        {
        emp      foreman[5];
        emp      engineer[5];
        for(int i=0;i<5;i ++)
        {
        cout<<" for foreman:";
        foreman[i] . getdata();
        }
        cout<<endl;
        for(i=0;i<5;i++)
        {
        Foreman[i].putdata(); .
        }
        for(int i=0;i<5;i ++)
        {
        cout<<" for engineer:";
        ingineer[i].getdata();
        }
        for(i=0;i<5;i++)
        {
        ingineer[i].putdata();
        }
        getch();
        return(0);
        }
```

REPLACE AND SORT USING CLASS:-

```
#include<iostream.h>
#include<constream.h>
        class sort
                {
        private:
                int nm[30];
        public;
                void getdata();
                void putdata();
        }:
void sort :: getdata()
        {
        int i,j,k;
        cout<<"enter 10 nos:" ;
        for(i=0;i<10;i++)
                {
                cin>>nm[i];
                }
        for(i=0;i<9;i++)
                {
                for(j=i+l:j<10:j++)
                {
                        if(nm[i]>nm[j])
                                {
                                k=nm[i];
                                nm[i]=nm[j];
                                nm[j]=k;
                                }
                }

void sort :: putdata()
        {
        int k;
        for(k=0;k<10;k++)
        {
                cout<<num [k] <<endl ;
```

```
}
}
int main()
{
clrscr();
sort s;
s.getdata();
s.putdata();
return(0);
}
```

ARRAY OF MEMBERS:

```
#include<iostream.h>
#include<constream.h>
    const int m=50;
    class items
    {
        int    item_code[m];
        floatitem_price[m];
        intcount;
    public:
        void cnt(void) { count=0;}
        void get_item(void);
        void display_sum(void);
        void remove(void);
        void display _item(void);
    };

    void items :: get_item (void)
        {
            cout<<"enter itemcode:";
            cin>> item_code[code];
            cout<<"enter item cost:";
            cin>>item_price[count];
            count ++ ;
        }

    void items :: display _sum(void)
        {
            float sum=0;
            for( int i=0;i<count;i++)
                {
```

```
                                        sum=sum+item_price[i];
                                        }
                        cout<< "\n total value:"<<sum<<endl;
        }
int main ( )
{
        items order;
        order.cnt();
        int x;
                do
                {
                cout<<"\nyou can do the following:";
                cout<<"enter appropriate no:";
                cout<<endl<<" 1 :add an item'';
                cout<<endl<<"2: display total value :";
                cout<<endl<<"3 : display an item";
                cout<<endl<<"4 :display all item:";
                cout<<endl<<"5 : quit:";
                cout<<endl<<endl<<"what is your option:";
                cin>>x;

                switch(x)
                {
                case 1: order.get_item(); break;
                case 2: order.display_sum(); break;
                cose 3: order.remove(); break;
                case 4: order.display_item();break;
                case 5: break;
                default : cout<<"error in input; try again";
                }
    } while(x!=5);
}
```

STATIC DATA MEMBER:

A data member of a class can be qualified as static . The properties of a static member variable are similar to that of a static variable. A static membervariable has contain specialcharacteristics.

Variable has contain special characteristics:-

1) It is initialized to zero when the first object of its class is created.No other initialization ispermitted.
2) Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects arecreated.
3) It is visible only with in the class but its life time is the entire program. Static variables are normally used to maintain values common to the entire class. For example a static data member can be used as a counter that records the occurrence of all theobjects.

int item :: count; // definition of static data member

Note that the type and scope of each static member variable must be defined outside the class definition .This is necessary because the static data members are stored separately rather than as a part of anobject.

Example:-

```
#include<iostream.h>
        class item
        {
                static int count; //count is static
                int number;
        public:
                void getdata(int a)
.                 {
                        number=a;
                        count++;
                  }
                 void getcount(void)
                  {
                        cout<<"count:";
                        cout<<count<<endl;
                   }
        };
int item :: count ; //count defined
        int main( )
        {
        item a,b,c;
        a.get_count( );
        b.get_count( );
        c.get_count( ):
        a.getdata():
        b.getdata();
```

```
        c.getdata( );
        cout«"after reading data : "«endl;
            a.get_count( );
            b.gel_count( );
            c.get count( );
    return(0);
    }
```

The output would be

```
    count:0
    count:0
    count:0
```

After reading data

```
    count: 3
    count:3
    count:3
```

The static Variable count is initialized to Zero when the objects created . The count is incremented whenever the data is read into an object. Since the data is read into objects three times the variable count is incremented three times. Because there is only one copy of count shared by all the three object, all the three output statements cause the value 3 to be displayed.

STATIC MEMBER FUNCTIONS:-

A member function that is declared static has following properties :-

1. Astaticfunctioncanhaveaccesstoonly otherstaticmembersdeclaredinthe same class.
2. A static member function can be called using the class name as follows:-
   class - name :: function -name;

Example:-

```
#include<iostream.h>
    class test
    {
        int code;
        static int count; // static member variable
    public:
        void set(void)
        {
        code=++count;
        }
        void showcode(void)
        {
        cout<<"object member : "<<code<<end;
        }
    static void showcount(void)
        { cout<<"count="<<count<<endl; }
        };
int test:: count;
int main()
{
```

```
test t1,t2;
t1.setcode( );
t2.setcode();
test :: showcount();          '
testt3;
t3.setcode( );
test:: showcount( );
t1.showcode( );
t2.showcode( );
t3.showcode( );
return(0);
```

output:- count : 2
count: 3
object number1
object number2
object number3

OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as A function argument. This can cone in two ways

1. A copy of the entire object is passed to thefunction.
2. Only the address of the object is transferred to thefunction

The first method is called pass-by-value. Since a copy of the object is passed to the function, any change made to the object inside the function do not effect the object used to call the function.
The second method is called pass-by-reference . When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the functions will reflect in the actual object .The pass by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Example:-

```
#include<iostream.h>
        class time
        {
                int hours;
                intminutes;
        public:
                void gettime(int h, int m)
                {
                                hours=h;
                                minutes=m;
                }
        void puttime(void)
                {
                cout<< hours<<"hours and:";

                cout<<minutes<<"minutes:"<<end;
                }
```

```
                void sum( time ,time);
        };
void time :: sum (timet1,time t2)       .
        {
        minutes=t1.minutes + t2.minutes;
        hours=minutes%60;
        minutes=minutes%60;
        hours=hours+t 1.hours+t2.hours;
        }

        int main()
        {
        time T1,T2,T3;
        T1.gettime(2,45);
        T2.gettime(3,30);
        T3.sum(T1,T2);
        cout<<"T1=";
        T1.puttime( );
        cout<<"T2=";
        T2.puttime( );
        cout<<"T3=";
        T3.puttime( );
        return(0);
        }
```

FRIENDLY FUNCTIONS:-

We know private members can not be accessed from outside the class. That is a non - member function can't have an access to the private data of a class. However there could be a case where two classes manager and scientist, have been defined we should like to use a function income-tax to operate on the objects of both theseclasses.

In such situations, c++ allows the common function lo be made friendly with both the classes , there by following the function to have access to the private data of these classes .Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the classes as shown below :

```
class ABC
{
---------
---------
public:
        --------
        ----------
        friend void xyz(void);
};
```

The function declaration should be preceded by the keyword friend , The function is defined else where in the program like a normal C ++ function . The function definition does not use their the keyword friend or the scope operator **::** . The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any no of classes. A friend function, as though not a member function , has full access rights to the private members of the class.

A friend function processes certain special characteristics:

a. It is not in the scope of the class to which it has been declared asfriend.
b. Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a member function without the help of anyobject.
c. Unlike memberfunctions.

Example:

```
#include<iostream.h>
        class sample
        {
                int a;
                int b;
        public:
                void setvalue( ) { a=25;b=40;}
                friend float mean( sample s);
        }
        float    mean (samples)
                {
                        return (float(s.a+s.b)/2.0);
                }
 int main ( )
    {
```

```
                    sample x;
                    x . setvalue( );
                    cout<<"mean value="<<mean(x)<<endl;
                    return(0);

                    }
```

output:
mean value : 32.5

A function friendly to two classes

```
#include<iostream.h>
        class abc;
        class xyz
        {
                int x;
        public:
                void setvalue(int x) { x-= I; }
                friend void max (xyz,abc);
        };
        class abc
        {
                int a;
        public:
                void setvalue( int i) {a=i; }
                friend void max(xyz,abc);
        };


        void max( xyz m, abc n)
        {
                if(m . x >= n.a)
                        cout<<m.x;
                else
                        cout<< n.a;
        }

        int main( )
        {
        abc j;
        j . setvalue( 10);
        xyz s;
        s.setvalue(20);
        max( s , j );
        return(0);
        }
```

SWAPPING PRIVATE DATA OF CLASSES:

```
#include<iostream.h>

        class class-2;
        class class-1
        {
```

```
        int value 1;
public:
        void indata( int a) { value=a; }
        void display(void) { cout<<value<<endl; }
        friend void exchange ( class-1 &, class-2 &);
};

class class-2
{
        int value2;
public:
        void indata( int a) { value2=a; }
        void display(void) { cout<<value2<<endl; }
        friend void exchange(class-l & , class-2 &);
        };
void exchange ( class-1 &x, class-2 &y)
        {
                int temp=x. value 1;
                x. value I=y.valuo2;
                y.value2=temp;
        }

        int main( )
        {
        class-1 c1;
        class-2 c2;
        c1.indata(l00);
        c2.indata(200);
        cout<<"values before exchange:"<<endl;
        c1.display( );
        c2.display( );
        exchange (c1,c2);
        cout<<"values after exchange :"<< endl;
        c1. display ( );
        c2. display ( );
        return(0);
        }
```

output:

values before exchange

100

200

values after exchange

200

100

PROGRAM FOR ILLUSTRATING THE USE OF FRIEND FUNCTION:

```
#include< iostream.h>
classaccount1;
classaccount2
{
private:
        int balance;
public:
account2( ) { balance=567; }
void showacc2( )
{
cout<<"balanceinaccount2 is:"<<balance<<endl;
friend int transfer (account2 &acc2, account1 &acc1,int amount);
};
class acount1
{
private:
        int balance;
public:
        account1 ( ) { balance=345; }


        void showacc1 ( )
        {
                cout<<"balance in account1 :"<<balance<<endl;
        }
friend int transfer (account2 &acc2, account1 &acc1 ,int amount);
        };

int transfer ( account2 &acc2, account1 & acc1, int amount)
        {
                if(amount <=accl . bvalance)
                        {
                        acc2. balance + = amount;
                        acc1 .balance - = amount;
                        }
                else
                        return(0);
        }
int main()
{
account1    aa;
account2    bb;




        cout << "balance in the accounts before transfer:" ;
        aa . showacc1();
        bb . showacc2();
        cout << "amt transferred from account1 to account2 is:";
        cout<<transfer ( bb,aa,100)<<endl;
```

```
                    cout<< " balance in the accounts after the transfer:";
                    aa . showacc 1 ( );
                    bb. showacc 2( );
                    return(0);
}
```

output:

balance in the accounts before transfer

balance in account 1 is 345

balance in account2 is 567

and transferred from account! to account2 is 100

balance in account 1 is245

balance in account2 is667

RETURNING OBJECTS:

```
# include< iostream,h>
        class complex
        {
                float x;
                float y;
        public:
                void input( float real , float imag)
                        {
                                x=real;
                                y=imag;
                        }
                friend complex sum( complex , complex);
                        void    show ( complex);
        };
complex sum ( complex c1, complex c2)
                {
                complex c3;
                c3.x=c1.x+c2.x;
                c3.y=c1.y+c2.y;
                return c3;}



                void complex :: show ( complex c)
                {
                cout<<c.x<<" +j "<<c.y<<endl;
                 }

                intmain( )
                {
                complex a, b,c;
                a.input(3.1,5.65);
                b.input(2.75,1.2);
                c=sum(a,b);
                cout <<" a="; a.show(a);
                cout <<" b= "; b.show(b);
                cout <<" c=" ; c.show(c);
                return(0);
                }
```

output:

```
a =3.1 + j 5.65
b= 2.75+ j 1.2
c= 5.55 + j 6.85
```

POINTER TO MEMBERS;

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name.

A class member pointer can be declared using the operator **::** * with the class name.

For Example:

```
classA
{
private:
        int m;
public:
        void show( );
};
```

We can define a pointer to the member m as follows :

int A **::** * ip = & A **::** m

The ip pointer created thus acts like a class member in that it must be invoked with a class object. In the above statement. The phrase A **::** * means "pointer - to - member of a class" . The phrase & A **::** m means the " Address of the m member of a class"

The following statement is not valid :

```
int *ip=&m ; // invalid
```

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer ip can now be used toaccessthe m inside the member function (or friendfunction).

Let us assume that "a" is an object of " A" declared in a member function . We can access "m" using the pointer ip as follows.

```
cout<< a . * ip;
cout<< a.m;
ap=&a;
cout<< ap-> * ip;
cout<<ap->a;
```

The deferencing operator ->* is used as to accept a member when we use pointers to both the object and the member. The dereferencing operator. .* is used when the object itself is used with the member pointer. Note that * ip is used like a member name.

We can also design pointers to member functions which ,then can be invoked using the deferencing operator in the main as shownbelow.

(object-name.* pointer-to-member function)

(pointer-to -object -> * pointer-to-memberfunction)

The precedence of ( ) is higher than that of .* and ->* , so the parenthesis are necessary.

DEREFERENCING OPERATOR:

```
#include<iostream.h>
        class M
        {
                int x;
                int y;
        public:
                void set_xy(int a,int b)
                        {
                                x=a;
                                y=b;
                        }
        friend int sum(M);
        };

        int sum (M m)
        {
                int M :: * px= &M :: x; //pointer to member x




                        int M :: * py- & m ::y;//pointer to y
                        M * pm=&m;
                        int s=m.* px + pm->py;
                        return(s);
                }
        int main ( )
                {
                M m;
                void(M::*pf)(int,int)=&M::set-xy;//pointer to function set-xy (n*pf)( 10,20);
                //invokes set-xy
                cout<<"sum=:"<<sum(n)<<cncil;
                n *op=&n; //point to object n
                ( op->* pf)(30,40); // invokes set-xy
                cout<<"sum="<<sum(n)<<end 1 ;
                return(0);
                }
output:
        sum= 30
        sum=70
```

CONSTRUCTOR:

A constructor is a special member function whose task is to initialize the objects of its class . It is special because its name is the same as the class name. The constructor is invoked when ever an object of its associated class is created. It is called constructor because it construct the values of data members of theclass.

A constructor is declared and defined as follows:

```
//'class with a constructor
        class integer
        {
                int m,n:
        public:
                integer! void);//constructor declared
                ------------
                ------------
        };
integer :: integer(void)
        {
                m=0;
                n=0;
        }
```

When a class contains a constructor like the one defined above it is guaranteed that an object created by the class will be initialized automatically.

For example:-

```
Integer int1; //object int 1 created
```

This declaration not only creates the object int1 of type integerbutalso initializes its data members m and n tozero.

A constructor that accept no parameter is called the default constructor. The default constructor for class A is A :: A( ). If no such constructor is defined, then the compiler supplies a default constructor.

Therefore a statement such as :-

```
A a ;//invokes the default constructor of thecompilerof the
  compiler to create the object "a";
```

Invokes the default constructor of the compiler to create the object a.

The constructor functions have some characteristics:-

- They should be declared in the public section.
- They are invoked automatically when the objects arecreated.
- They don't have return types, not even void and therefore they cannot returnvalues.
- They cannot be inherited , though a derived class cancall

the base class constructor .

- Like other C++ function , they can have defaultarguments,
- Constructor can't bevirtual.
- An object with a constructor can't be used as a member of union.

Example of default constructor:

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

        class abc
        {
        private:
                char nm[];
        public:
                abc ( )
                {
                        cout<<"enter your name:";
                        cin>>nm;
                }
                void display( )

                        {
                                cout<<nm;
                        }

                        };

                        int main( )
                        {
                        clrscr( );
                        abc d;
                        d.display();
                        getch( );
                        return(0);
                        }
```

PARAMETERIZED CONSTRUCTOR:-

the constructors that can take arguments are called parameterized constructors. Using parameterized constructor we can initialize the various data elements of different objects with different values when they are created.

Example:-

```
        class integer
        {
                int m,n;
        public:
                integer( int x, int y);
                        --------
                        ---------
                };
```

```
integer:: integer (int x, int y)
                {
                        m=x;n=y;
                }
```

the argument can be passed to the constructor by calling the constructor implicitly.

```
integer int 1 = integer(0,100); // explicit call
integerint1(0,100);       //implicitecall
```

CLASS WITH CONSTRUCTOR:-

```
#include<iostream.h>
class integer
{
        int m,n;
public:
        integer(int,int);
        void display(void)




        {
                cout<<"m=:"<<m ;
                cout<<"n="<<n;
        }
};
        integer :: integer( int x,int y) // constructor defined
                {
                m=x;
                n=y;
                }
        int main( )
        {
                integer int1(0,100);      // implicit call
                integerint2=integer(25,75);
                cout<<" \nobjectl"<<endl;
                        int1.display();
                        cout<<" \n object2 "<<endl;
                        int2.display();
        }
```

output:

```
        object 1
        m=0
        n=100
        object2
        m=25
        n=25
```

Example:-

```
#include<iostream.h>
#include<conio.h>
class abc
{
private:
        char nm [30];
        int age;
public:
        abc ( ){ }// default
        abc ( char x[], int y);
        void get( )
        {
        cout<<"enter your name:";
        cin>>nm;
        cout<<" enter your age:";
        cin>>age;
        }
void display( )
{
        cout<<nm«endl;
        cout«age;
}
};
        abc : : abc(char x[], int y)
                {
                        strcpy(nm,x);
                        age=y;
                }
void main( )
{
abc 1;
abc m=abc("computer",20000);
l.get();
l.dispalay( );
m.d isplay ();
getch( );
}
```

OVERLOADED CONSTRUCTOR:-

```
#include<iostream.h>
#include<conio.h>
        class sum
        {
        private;
                int a;
                int b;
                int c;
                float d;
                double e;
        public:
                sum ( )
```

```
{
cout<<"enter a;";
cin>>a;
cout<<"enter b;";
cin>>b;
cout<<"sum= "<<a+b<<endl;
        }
        sum(int a,int b);
        sum(int a, float d,double c);
        };
        sum :: sum(int x,int y)
                {
                        a=x;
                        b=y;
                }
        sum :: sum(int p, float q ,double r)
                {
                        a=p;
                        d=q;
                        e=r;
                }
        void main( )
        {
        clrscr( );
        sum 1;
        sum m=sum(20,50);
        sum n= sum(3,3.2,4.55);
        getch();
        }

        output:
        enter a : 3
        enter b : 8
        sum=11
        sum=70
        sum=10.75
```

COPY CONSTRUCTOR:

A copy constructor is used to declare and initialize an object from another object.

Example:-

the statement

integer 12(11);

would define the object 12 and at the same time initialize it to the values of 11.

Another form of this statement is : integer 12=11;

The process of initialization through a copy constructor is known as copy initialization.

Example:-

```
#incliide<iostream.h>
        class code
        {
                int id;
```

```
public
        code ( ) { } //constructor
        code (int a) { id=a; } //constructor
        code(code &x)
        {
                Id=x.id;
        }
        void display( )
        {
        cout<<id;
        }
};
int main( )
{
code A(100);
code B(A);
code C=A;
code D;
D=A;
cout<<" \n id of A :"; A.display( );
cout<<" \nid of B :"; B.display( );
cout<<" \n id of C:"; C.display( );
cout<<" \n id of D:"; D.display( );
}
```

output :-

```
id of A:100
id of B:100
id of C:100
id ofD:100
```

DYNAMICCONSTRUCTOR:-

The constructors can also be used to allocate memory while creating objects . This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.

Allocate of memory to objects at the time of their construction is known as dynamic constructors of objects. The memory is allocated with the help of new operator.

Example:-

```
#include<iostream.h>
#include<string.h>
class string
{
        char *name;

        int    length;
public:
        string ()
```

```
        {
        length=0;
        name= new char [length+1]; /* one extra for \0 */
        }
string( char *s) //constructor 2
        {
        length=strlen(s);
        name=new char[length+1];
        strcpy(name,s);
        }
void display(void)
{
        cout<<name<<endl;
}
void join(string &a .string &b)
        {
        length=a. length +b . length;
        delete name;
        name=new char[length+l]; /* dynamic allocation */
        strcpy(name,a.name);
        strcat(name,b.name);
        }
        };
int  main()
{
char * first = “Joseph” ;
string name1(first),name2(“louis”),naine3( “LaGrange”),sl,s2;
sl.join(name1,name2);
s2.join(s1,name3);
namel.display( );
name2.display( );
name3.display( );
s1.display();
s2.display();
}
output :-
        Joseph
        Louis
        language
        Joseph Louis
        Joseph Louis Language
```

DESTRUCTOR:-

A destructor, us the name implies is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by atilde.

ForExample:-

~ integer( ) { }

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructor in a program since it releases memory space for future use.

Delete is used to free memory which is created by new.

Example:-

```
matrix : : ~ matrix( )
{
        for(int i=0; i<11;i++)
                delete p[i];
                delete p;
}
```

IMPLEMENTED OFDESTRUCTORS:-

```
#include<iostream.h>
        int count=0;
        class alpha
        {
        public:
                alpha( )
                {
                count ++;
                cout<<"\n no of object created :"<<endl;
                }
                ~alpha( )
                {
                        cout<<"\n no of object destroyed :" <<endl;
                        coutnt--;
                }
        };



int  main()
{

cout<<" \n \n enter main \n:";
alpha A1,A2,A3,A4;
{
        cout<<" \n enter block 1 :\n";
```

```
                alpha A5;
        }
        {
        cout<<" \n \n enter block2 \n";
        alphaA6;
        }
cout<<\n re-enter main \n:";
        return(0);
}

        output:-
enter main
no of object created 1
no of object created 2
no of object created 3
no of object created 4
enter block1
no of object created 5
no of object destroyed 5
enter block2
no of object created 5
no of object destroyed 5
re-enter main
no of object destroyed 4
no of object created 3
no of object created 2
no of object created1

Example :-
        #include<iostream.h>
        int x=l;
        class abc
        {
        public:
                abc( )
                {
                        x--;
                        cout<<"construct the no"<<x<<endl;
                }
                ~abc( )
                {
                cout<<"destruct the no:"<<x<<endl;
                        x--;
                                }
                                };
                        int main( )
                        {
                        abc I1,I2,I3,I4;
                        cout«ll«12«13«l4«endl;
                        return(0);
                        }
```

OPERATOR OVERLOADING:-

Operator overloading provides a flexible option for the creation of new definations for most of the C++ operators. We can overload all the C++ operators except the following:

- Class members access operator (. ,.*)
- Scope resolution operator (::)
- Size operator(sizeof)
- Condition operator (?:)

Although the semantics of an operator can be extended, we can't change its syntax, the grammatical rules that govern its use such as the no of operands precedence and associativety. For example the multiplication operator will enjoy higher precedence than the addition operator.

When an operator is overloaded, its original meaning is not lost. For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

DEFINING OPERATOR OVERLOADING:

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied . This is done with the help of a special function called operator function, which describes the task.

Syntax:-

```
return-type class-name :: operator op( arg-list)
        {
            function body
        }
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator, operator op is the function name.

operator functions must be either member function, or friend function. A basic defference between them is that a friend function will have only one argument for unary operators and two for binary operators, This is because the object used to invoke the member function is passed implicitly and therefore is available for the member functions. Arguments may be either by value or by reference.

operator functions are declared in. the class using prototypes as follows:-

```
vector operator + (vector); /./ vector addition
vector operator-( ); //unary minus
friend vector operator + (vuelor, vector); // vector add
friend vectoroperator -(vector);      // unary minus
vector operator - (vector&a);         //substraction
int operator==(vector);        //comparision
friend int operator = =(vector ,vrctor); // comparision
```

vector is a data type of class and may represent both magnitude and direction or a series of points called elements.

The process of overloading involves the following steps:-

1. Create a class that defines the data type that is used in the overloadingoperation.
2. Declare the operator function operator op() in the public part of theclass
3. It may be either a member function or friendfunction.
4. Define the operator function to implement the requiredoperations.

Overloaded operator functions can be invoked by expressions such as

op x or x op;

for unary operators and

x op y

for binary opearators.

operator op(x);

for unary operator using friend function

operator op(x,y);

for binary operator usinf friend function.

Unary – operator overloading(using member function):

```
class abc
{
int m,n;
public:
abc()
{
m=8;
n=9;
}
void show()
{
cout<<m<<n;
}
operator --()
{
--m;
--n;
}
};
void main()
{
abc x;
x.show();
--x;
```

```
x.show();
}
```

Unary – - operator overloading(using friend function):

```
class abc
{
int m,n;
public:
abc()
{
m=8;
n=9;
}
void show()
{
cout<<m<<n;
}
friend operator --(abc &p);
};
operator -- (abc &p)
{
--p.m;
--p.n;
}
};
void main()
{
abc x;
x.show();
operator--(x);
x.show();
}
```

Unary operator+ for adding two complex numbers (using member function)

```
class complex
{
float real,img;
public:
complex()
{
        real=0;
        img=0;
}
complex(float r,float i)
{
real=r;
img=i;
}
void show()
{
cout<<real<<"+i"<<img;
}
complex operator+(complex &p)
{
        complex w;
        w.real=real+q.real;
        w.img=img+q.img;
        return w;
}
};
void main()
{
complex s(3,4);
complex t(4,5);
complex m;
m=s+t;
s.show();
t.show();
m.show();
}
```

Unary operator+ for adding two complex numbers (using friend function)

```
class complex
{
float real,img;
public:
complex()
{
        real=0;
        img=0;
}
complex(float r,float i)
{
real=r;
img=i;
```

```
}
void show()
{
cout<<real<<"+i"<<img;
}
friend complex operator+(complex &p,complex &q);
};
complex operator+(complex &p,complex &q)
{
        complex w;
        w.real=p.real+q.real;
        w.img=p.img+q.img;
        return w;
}
};
voidmain()
{
complex s(3,4);complex t(4,5);
complexm;
m=operator+(s,t);
s.show();t.show();
m.show();
}
```

Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```
class integer
{
intx, y;
        public:
        intoperator + ( ) ;
}
int integer: : operator + ( )
{
        return (x-y) ;
}
```

Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).

Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Table 7.2

Operator to Overload	Arguments passed to the Member Function	Arguments passed to the Friend Function
Unary Operator	**No**	**1**
Binary Operator	**1**	**2**

Type Conversions

In a mixed expression constants and variables are of different data types. The assignment operations causes automatic type conversion between the operand as per certain rules.

The type of data to the right of an assignment operator is automatically converted to the data type of variable on the left.

Consider the followingexample:

```
intx;
float y = 20.123;
x=y;
```

This converts float variable y to an integer before its value assigned to x. The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator in case of objects to copy values of all data members of right hand object to the object on left hand. The objects in this case are of same data type. But of objects are of different data types we must apply conversion rules for assignment.

There are three types of situations that arise where data conversion are between incompatible types.

1. Conversion from built in type to classtype.
2. Conversion from class type to built intype.
3. Conversion from one class type toanother.

Basic to Class Type

A constructor was used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. In these examples constructors performed a defacto type conversion from the argument's type to the constructor's class type

Consider the following constructor:

```
string :: string (char*a)
{
        length = strlen (a);
        name=new char[len+1];
        strcpy (name,a);
}
```

This constructor builds a string type object from a char* type variable a. The variables length and name are data members of the class string. Once you define the
constructor in the class string, it can be used for conversion from char* type to string type.

Example

```
string si , s2;
char* namel = “Good Morning”;
char* name2 = “ STUDENTS” ;
s1 = string(namel);
s2 = name2;
```

The program statement

```
si = string (namel);
```

first converts name 1 from char* type to string type and then assigns the string type values to the object s1. The statement

```
s2 = name2;
```

performs the same job by invoking the constructor implicitly.
Consider the following example

```
class time
{
        int hours;
        intminutes;
                public:
        time (int t) // constructor
        {
                hours = t  /  60;                  //t is inputted in minutes
                minutes = t %60;
                }
        };
```

In the following conversion statements :

```
timeTl;                 //object  Tl  created
int period =160;
Tl= period;             //int to classtype
```

The object Tl is created. The variable period of data type integer is converted into class type time by invoking the constructor. After this conversion, the data member hours ofTl will have value 2 arid minutes will have a value of 40 denoting 2 hours and 40 minutes.

Note that the constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded =operator.

Class to Basic Type

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overloaded casting operator that convert a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename ( )
{
        //Program statmerit .
}
```

This function converts a class type data to typename. For example, the operator double( ) converts a class object to type double, in the following conversion function:

```
vector:: operator double ( )
{
        double sum = 0 ;
        for(int I = 0; ioize;
        sum =  sum + v[i]   * v[i ];      //scalar magnitude
        returnsqrt(sum);
}
```

The casting operator should satisfy the following conditions.

- It must be a classmember.
- It must not specify a return type.
- It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion inside the function belongs to the object that invoked the function. As a result function does not need anargument.

In the string example discussed earlier, we can convert the object string to char* as follows:

```
string:: operator char*( )
{
        return (str) ;
}
```

One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type.

Example

Obj1 = Obj2 ; //Obj1 and Obj2 are objects of different classes.

Objl is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Objl. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destinationclass.

We studied that the casting operator function

Operator typename( )

Converts the class object of which it is a member to typename. The type name may be a built-in type or a user defined one(another class type) . In the case of conversions between objects,

typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used. The conversion takes place in the source class and the result is given to the destination class object.

Let us consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. The argument belongs to the source class and is passed to the destination class for conversion. Therefore the conversion constructor must be placed in the destinationclass.

Table 7.3

Conversion	**Conversion takes place in**	
	Source class	**Destination class**
Basic to class	Not applicable	Constructor
Class to Basic	Casting operator	Not applicable
Class to class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destinationclass.

Consider the following example of an inventory of products in a store. One way of keeping record of the details of the products is to record their code number, total items in the stock and the cost of each item. Alternatively we could just specify the item code and the value of the item in the stock. The following program uses classes and shows how to convert data of one type to another.

```
#include<iostream.h>
#include<conio.h>
class stock2;
class stock1
{
int code, item;
float price;
public:
stockl (int a, int b, float c)
{
code=a;
item=b;
price=c;
}
void disp( )
{
cout<<"code"<<code <<"\n";
cout<<"Items"<<item <<"\n";
cout<<"Price per item Rs . "<<price <<"\n";
}
int getcode()
{return code; }
int getitem()
{return item; }
int getprice( )
{return price;}
```

```
operator float( )
{
return ( item*price );
}
};

class stock2
{
int  code;
float val;
public:
stock2()
{
code=0; val=0;
}
stock2(int x, float y)
{
code=x; val=y;
}
void disp( )
{
cout<< “code”<<code << “\n”;
cout<< “Total Value Rs . “ <<val<<”\n”
}
stock2 (stockl p)
{
code=p . getcode ( ) ;
val=p.getitem( ) * p. getprice ( ) ;
}
};

void main ( )
{          '
Stockl il(101, 10,125.0);
Stock2 12;
float tot_val;
tot_val=i1 ;
i2=il;
cout<<” Stock Details-stockl-type” <<”\n”;
i 1 . disp ( ) ;
cout<<” Stock value”<<”\n”;
cout<< tot_val<<”\n”;
cout<<” Stock Details-stock2-type”<< “\n”;
i2 .disp( );
getch ( ) ;
}
```

You should get the following output.

Stock Details-stock1-type

code 101

Items 10

Price per item Rs. 125

Stock value

1250

Stock Details-stock2-type
code 10 1
Total Value Rs. 1250

UNIT-III

Inheritance:

Reaccessability is yet another feature of OOP's. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERTTENCE'. This is often referred to as IS-A' relationship because very object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class and thenew one iscalled'DERIEVED'class.

Defining DerivedClasses

A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class d_classname : Access specifier baseclass name
{
__
___// members of derivedclass
};
```

The colon indicates that the a-class name is derived from the base class name. The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the status of derived features e.g.

```
class xyz        //baseclass
{
        members of xyz
};
class ABC :publicxyz      //publicderivation
{
        members of ABC
};
class ABC: XYZ     //private derivation (bydefault)
{
        members of ABC
};
```

In the 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. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Single Inheritance

When a class inherits from a single base class, it is known as single inheritance. Following program shows the single inheritance using public derivation.

```
#include<iostream.h>
#include<conio.h>
class worker
{
```

```
        int age;
        char name [10];
        public:
        void get ( );
};
void worker : : get ( )
{
        cout <<"yout name please"
        cin >> name;
        cout <<"your age please" ;
        cin >> age;
}
void worker :: show ( )
{
        cout <<"In My name is :"<<name<<"In My age is :"<<age;
}
class manager ::publicworker      //derived class(publicly)
{
        int now;
        public:
        void get ( ) ;
        void show ( ) ;
};
void manager : : get ( )
{
        worker : : get ();      //the calling of base class input fn.
        cout << "number of workers underyou";
        cin >> now;
        cin>>name>>age;
}                               ( if they were public )
void manager :: show ()
{
        worker :: show();       //calling of base class o/pfn.
        cout <<"in No. of workers under me are: " << now;
}
        main ( )
{
        clrscr ( ) ;
        worker W1;
        manager M1;
        M1 .get ( );
        M1.show ( ) ;
}
```

If you input the following to this program:

Your name please

Ravinder

Your age please

27

number of workers under you

30

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
classworker     //Base classdeclaration
{
        int age;
        char name [10] ;
        public:
        void get ( ) ;
        void show ( ) ;
};
void worker : : get ( )
{
        cout << “your name please” ;
        cin >> name;
        cout << “your age please”;
        cin >>age;
}
void worker : show ( )
{
        cout << “in my name is: “ <<name<< “in” << “my age is : “ <<age;
}
class manager : worker //Derived class (privately by default)
{
        int now;
        public:
        void get ( ) ;
        void show ( ) ;
};
void manager : : get ( )
{
        worker : : get ( ); //calling the get function of base
        cout << “number of worker under you”; class which is
        cin >> now;
        }
void manager : : show ( )
{
        worker : : show ( ) ;
        cout << “in no. of worker under me are : “ <<now;
}
main ( )
{
```

```
        clrscr ( ) ;
        worker wl ;
        manager ml;
        ml.get ( ) ;
        ml.show ( );
}
```

The following program shows the single inheritance using protected derivation

```
#include<conio.h>
#include<iostream.h>
classworker             //Base class declaration
{ protected:
        int age; char name [20];
        public:
        void get ( );
        void show ( );
};
void worker :: get ()
{
        cout >> “your name please”;
        cin >>name;
        cout << “your age please”;
        cin >> age;
}
void worker :: show ( )
{
        cout << “in my name is: “ << name << “in my age is “ <<age;
}
class manager:: protected worker // protected inheritance
{
        int now;
        public:
        void get ();
        void show ( ) ;
};
void manager : : get ( )
{
        cout << “please enter the name In”;
        cin >> name;
        cout<< “please enter the age In”; //Directly inputting thedata
        cin>>age;       members of baseclass
        cout << “ please enter the no. of workers under you:”;
        cin >> now;
}
void manager : : show ( )

{
        cout « "your name is : "«name«" and age is : "«age;
        cout «"In no. of workers under your are : "«now;
main ( )
{
        clrscr ( ) ;
        manager ml;
        ml.get ( ) ;
```

```
        cout « "\n \n";
        ml.show ( );
}
```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

a. **Private:** when a base class is 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 be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derivedclass.

b. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derivedclass.

c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these twoclasses.

The below mentioned table summarizes how the visibility of members undergo modifications when they are inherited

Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

The private and protected members of a class can be accessed by:

a. A function i.e. friend of aclass.

b. A member function of a class that is the friend of theclass.

c. A member function of a derived class.

Student Activity

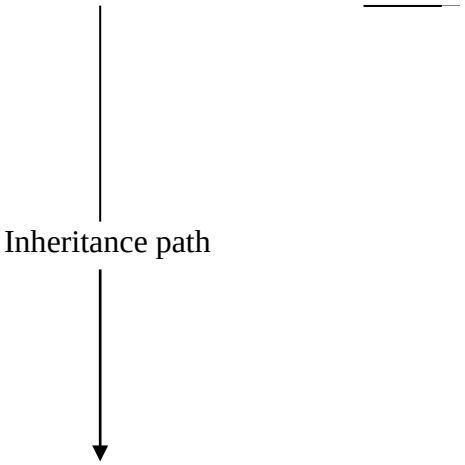
1. Define Inheritance. What is the inheritance mechanism inC++?

2. What are the advantage ofInheritance?

3. What should be the structure of a class when it has to be a base for otherclasses?

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called ‘MULTILEVEL INHERITENCE’. The class B is known as the ‘INTERMEDIATE BASE CLASS’ since it provides a link for the inheritance between A and C. The chain ABC is called ‘ITNHERITENCE*PATH’ for e.g.



The declaration for the same would be:

```
Class A
{
//body
}
Class B : public A
{
//body
}
Class C : public B
{
//body
}
```

This declaration will form the different levels of inheritance.

Following program exhibits the multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
classworker       // Base classdeclaration
{
        int age;
        char name [20] ;
        public;
        void get( ) ;
```

```
        void show( ) ;
}



void worker: get ( )
{
        cout << “your name please” ;
        cin >> name;
        cout << “your age please” ;
}
void worker : : show ( )
{
        cout << “In my name is : “ <<name<< “ In my age is : “ <<age;
}
class manager : public worker //Intermediate base class derived
{                   //publicly from the base class
        intnow;
        public:
        void get ( ) ;
        void show( ) ;
};
void manager :: get ( )
{
        worker : :get ();       //calling get ( ) fn. of base class
        cout << “no. of workers under you:”;
        cin >> now;
}
void manager : : show ( )
{
        worker : : show ();     //calling show ( ) fn. of base class
        cout << “In no. of workers under me are: “<<now;
}
class ceo:publicmanager       //declaration of derivedclass
{                   //publicly inherited fromthe
        intnom;         //intermediate baseclass
        public:
        void get ( ) ;
        void show ( ) ;
};
void ceo : : get ( )
{
        manager : : get ( ) ;
        cout << “no. of managers under you are:”; cin >> nom;
}
void manager : : show ( )
{
        cout << “In the no. of managers under me are: In”;
        cout << “nom;
}
```

```
main ( )
{
        clrscr ( ) ;
                ceo cl ;
                cl.get ( ) ; cout << "\n\n";
                cl.show ( ) ;
        }
```

Worker

Private: int age; char name[20];
Protected:
Private: int age; char name[20];

Manager:Worker

Private: int now;
Protected:
Public: void get() void show() worker::get() worker::get()

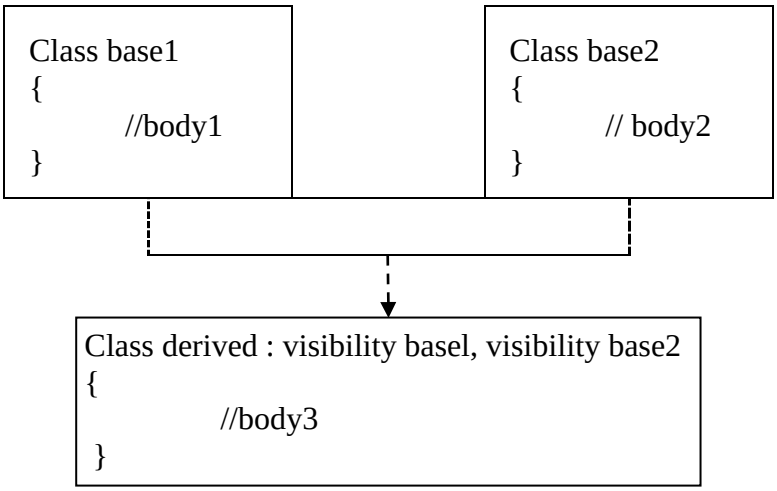
Ceo: Manager

Public:
Protected:
Public: All the inherited members

Multiple Inheritances

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features

of several existing classes as a starring point for defining new classes. It is like the child inheriting the physical feature of one parent and the intelligence of another. The syntax of the derived class is asfollows:

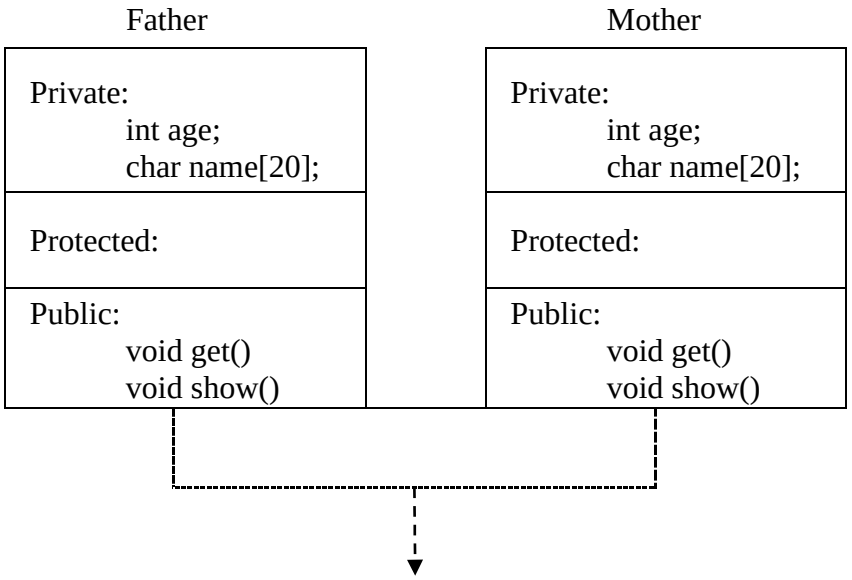


Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multipleinheritance.

```
#include<iostream.h>
#include<conio . h>
classfather                 //Declaration of baseclassl
{
        int age ;
        char flame [20] ;
        public:
        void get ( ) ;
        void show ( ) ;
};
void father : : get ( )
{
        cout << "your father name please";
        cin >> name;
        cout << "Enter the age";
        cin >> age;
}
void father : : show ( )
{
cout<< "In my father's name is: ' <<name<< "In my father's age is:<<age;
}
classmother                         //Declaration of base class 2
{
        char name [20] ;
        int age ;
```

```
        public:
        void get ( )
        {
                cout << “mother’s name please” << “In”;
                cin >> name;
                cout << “mother’s age please” << “in”;
                cin >> age;
        }
void show ( )
{
        cout << “In my mother’s name is: “ <<name;
        cout << “In my mother’s age is: “ <<age;
        }
        class daughter : public father, public mother //derived class inheriting
        {                               //publicly
        char name[20];      //the features of both the baseclass
        intstd;
        public:
                void get ( ) ;
                void show ( ) ;
        };
        void daughter :: get ( )
        {
                father :: get ( ) ;
                mother :: get ( ) ;
                cout << “child's name: “;
                cin >> name;
                cout << “child's standard”;
                cin >> std;
        }
        void daughter :: show ()
        {
                father :: show ();
                nfather :: show ( ) ;
                cout << “In child’s name is : “ <<name;
                cout << “In child's standard: “ << std;
        }
        main ( )
        {
        clrscr ( ) ;
                daughter d1;
                d1.get ( ) ;
                d1.show ( ) ;
        }
```

Diagrammatic Representation of Multiple Inheritance is asfollows:



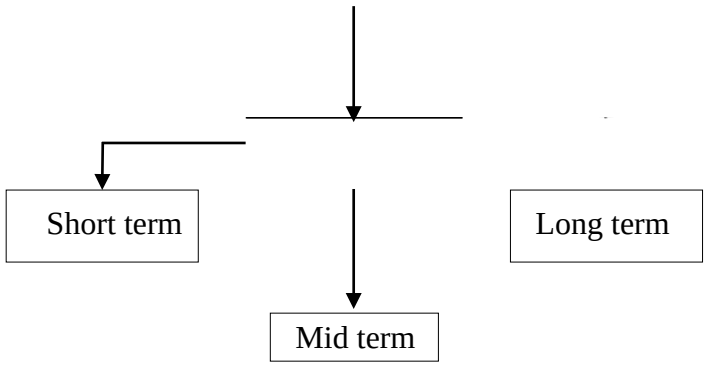
Class daughter: public Father, publicMother

Private: char name[20]; int age;
Protected:

Public:
//self
void get(); void showQ;
//from Father
void get(); void show();
//from Mother
void get(); void show();

Hierarchical Inheritance

Another interesting application of inheritance is to use is as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.



In general the syntax is given as

In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

```
// Program to show the hierarchical inheritance
#include<iostream.h>
# include<conio. h>
classfather                 //Base classdeclaration
{
        int age;
        char name [15];
        public:
        void get ( )
        {
                cout<< "father name please"; cin >> name;
```

```
                cout<< “father’s age please”; cin >> age;
        }
        void show ( )
        {
                cout << “In father’s name is ‘: “<<name;
                cout << “In father’s age is: “<< age;
        }
};
class son :publicfather          //derived class1
{
        char name [20] ;
        int age ;
        public;
        void get ( ) ;
        void show ( ) ;
} ;
void son : : get ( )




{
        father :: get ( ) ;
        cout << “your (son) name please” << “in”; cin >>name;
        cout << “your age please” << “ln”; cin>>age;
}
void son :: show ( )
{
        father : : show ( ) ;
        cout << “In my name is : “ <<name;
        cout << “In my age is : “ <<age;
}
class daughter :publicfather             //derived class2.
{
        char name [15] ;
        int age;
        public:
        void get ( )
        {
                father : : get ( ) ;
                cout << “your (daughter’s) name please In” cin>>name;
                cout << “your age please In”; cin >>age;
        }
        void show ( )
        {
                father : : show ( ) ;
                cout << “in my father name is: “ << name << “
                In and his age is : “<<age;
        }
};
main ( )
{
        clrscr ( ) ;
```

```
        son S1;
        daughter D1 ;
        S1. get ( );
        D1. get ( ) ;
        S1 .show( ) ;
        D1. show ( ) ;
}
```

Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

```
//Program to show the simple hybridinheritance
    #include<i sos t ream. h>
    #include<conio .h>
    classstudent            //base classdeclaration
    {
        protected:
        int r_no;
            public:
                void get _n (int a)
                {
                    r_no =a;
                    }
                void put_n (void)
                {
                cout << "Roll No. : "<< r_no;
                cout << "In";
                }
            };
            class test : public student
            {           //Intermediate baseclass
            protected : int parti, par2;

            public :
                void get_m (int x, int y) {
                    parti = x; part 2 = y; }
                void put_m (void) {
                    cout << "marks obtained: " << "In"
                        << "Part 1 = " << part1 << "in"
                        << "Part 2 = " << part2 << "In";
                }
            };
            classsports             // base forresult
            {
            protected : int score;
            public:
                void get_s (int s){
                    score = s }
                void put_s (void){
                    cout << " sports wt. : " << score << "\n\n";
```

```
                    }
            };
        class result : public test, public sports //Derived from test
                    &sports
        {
            int total;
            public:
            void display (void);
            };


        void result : : display (void)
        {
            total = part1 + part2 + score;
            put_n ( ) ;.
            put_m   (  );
            put_S    ( );
            cout << "Total score: " <<total<< "\n"
}
main ( )
{
        clrscr ( ) ;
        result S1;
        S1.get_n (347) ;
        S1.get_m (30, 35);
        S1.get_s (7) ;
        S1.dciplay ( ) ;
}
```

Student Activity

1. What is the major use of multilevelInheritance?
2. How are arguments sent to the base constructors in multiple inheritance? Whose responsibility isit.
3. What is the difference between hierarchical and hybridInheritance.

Virtual Base Classes

We have just discussed a situation which would require the use of both multiple and multi level inheritance. Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes ‘parent1’ and ‘parent2’ which themselves has a common base class ‘grandparent’. The child inherits the traits of ‘grandparent’ via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as ‘INDIRECT BASE CLASS’. Now, the inheritance by the child might cause some problems. All the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via ‘parent2’. So, there occurs a duplicacy which should beavoided.

The duplication of the inherited members can be avoided by making common base class as the virtual base class: fore.g.

```
classg_parent
{
        //Body
};
class parent1: virtual public g_parent
{
        // Body
};

class parent2: public virtual g_parent
{
        // Body
};
class child : public parent1, public parent2
{
        // body
};
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class. Note that keywords ‘virtual’ and ‘public’ can be used in eitherorder.

//Program to show the virtual base class

```
#include<iostream.h>
#include<conio . h>
classstudent                // Base classdeclaration
{
        protected:
        int r_no;
        public:
        void get_n (inta)
        { r_no = a; }
        void put_n(void)
        { cout << “Roll No. “ << r_no<< “ln”;}
        };
```

```
class test : virtual public student // Virtually declaredcommon
{                       //base class 1
        protected:
        int part1;
        int part2;
        public:
        void get_m (int x, int y)
        { part1= x; part2=y;}
                void putm (void)
                {
                cout << "marks obtained: " << "\n";
                cout << "part1 = " << part1 << "\n";
                cout << "part2 = "<< part2 << "\n";
        }
};
class sports : public virtual student // virtually declared common
{                       //base class 2
        protected:
        int score;
                public:
                void get_s (int a) {
                score = a ;
                }
                void put_s (void)
                { cout << "sports wt.: " <<score<< "\n";}
                };
                class result: public test,publicsports        //derivedclass
                {
                        private : int total ;
                        public:
                                void show (void) ;
                };
                void result : : show (void)
                { total = part1 + part2 + score ;
                        put_n ( );
                        put_m ( );
                        put_s ( ) ; cout << "\n total score= " <<total<< "\n" ;
                }
                main ( )
                {
                        clrscr ( ) ;
                        result S1 ;
                        S1.get_n (345)
                        S1.get_m (30, 35) ;
                        S1.get-S (7) ;
                        S1. show ( );
                        }

        //Program to show hybrid inheritance using virtual base classes
                #include<iostream.h>
                #include<conio.h>
                Class A
                {
```

```
        protected:
                int x;
        public:
                void get (int) ;
                void show (void) ;
        };
                void A : : get (int a)
                        { x = a ; }
                void A : : show(void)
                        { cout << X ;}
                Class A1 : Virtual PublicA
                {
```

```
protected:
        int y ;
public:
        void get (int) ;
        void show (void);
};
void A1 :: get (int a)
        { y = a;}
void A1 :: show (void)
{
cout <<y ;
{
class A2 : Virtual public A
{
protected:
        int z ;
public:
        void get (int a)
        { z =a;}
        void show (void)
        { cout << z;}
};
class A12 : public A1, public A2
{
int r, t ;
public:
        void get (int a)
        { r = a;}
        void show (void)
        { t = x + y + z + r ;
                cout << "result =" << t ;
        }
};
main ( )
{
clrscr ( ) ;
```

```
        A12 r ;
        r.A : : get (3) ;
        r.A1 : : get (4) ;
        r.A2 : : get (5) ;
        r.get (6) ;
        r . show ( ) ;
}
```

Pointer:

Introduction

When an object is created from its class, the member variables and member functions are allocated memory spaces. The memory spaces have unique addresses. Pointer is a mechanism to access these memory locations using their address rather than the name assigned to them. You will study the implications and applications of this mechanism in detail in this chapter.

Pointer is a variable which can hold the address of a memory location rather than the value at the location. Consider the following statement

```
int num =84;
```

This statement instructs the compiler to reserve a 2-byte of memory location and puts the value 84 in that location. Assume that the compiler allocates memory location 1001 to num. Diagrammatically, the allocation can be shown as:

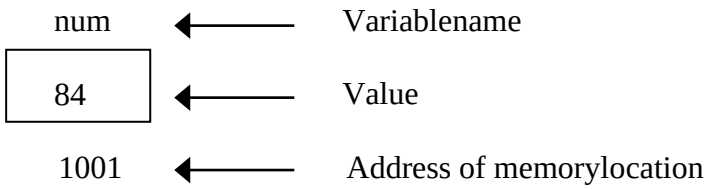


Figure 9.1

As the memory addresses are themselves numbers, they can be assigned to some other variable For example, ptr be the variable to hold the address of variable num.

Thus, we can access the value of num by the variable ptr. We can say “ptr points to num” as shown in the figurebelow.

Fig 9.2

Pointers to Objects

An object of a class behaves identically as any other variable. Just as pointers can be defined in case of base C++ variables so also pointers can be defined for an object type. To create a pointer variable for the following class

```
class employee{
        int code;
        char name [20] ;
      public:
        inline void getdata ( )= 0 ;
        inline void display ( )= 0 ;
};
```

The following codes is written

```
employee *abc;
```

This declaration creates a pointer variable abc that can point to any object of employee type.

this Pointer

C++ uses a unique keyword called "this" to represent an object that invokes a member function. 'this' is a pointer that points to the object for which this function was called. This unique pointer is called and it passes to the member function automatically. The pointer this acts as an implicit argument to all the member function, fore.g.

```
class ABC
{
  int a ;
        -----
        -----
 };
```

The private variable ‘a’ can be used directly inside a member function, like

```
a=123;
```

We can also use the following statement to do the same job.

```
this → a = 123
```

e.g.

```
class stud
{
        int a;
        public:
        void set (int a)
        {
        this → a = a; //here this point is used to assign a class level
        }       ‘a’ with the argument ‘a’
        void show ( )
        {
        cout << a;
        }
};
main ( )
{
        stud S1, S2;
```

```
        S1.bet (5) ;
        S2.show ();
}
o/p = 5
```

Pointers to Derived Classes

Polymorphism is also accomplished using pointers in C++. It allows a pointer in a base class to point to either a base class object or to any derived class object. We can have the following Program segment show how we can assign a pointer to point to the object of the derived class.

```
class base
{
        //Data Members
        //Member Functions
};
class derived : public base
{
        //Data Members
        //Member functions
};

void main ( ) {
        base *ptr; //pointer to class base
        derived obj ;
        ptr =&obj;        //indirect reference obj to thepointer
        //Other Program statements

}
```

The pointer ptr points to an object of the derived class obj. But, a pointer to a derived class object may not point to a base class object without explicit casting.

For example, the following assignment statements are not valid

```
void main ( )
{
        base obja;
        derived *ptr;
        ptr = &obja; //invalid.... .explicit casting required
        //Other Program statements
}
```

A derived class pointer cannot point to base class objects. But, it is possible by using explicit casting.

```
void main ( )
{
        base obj ;
        derived*ptr;            // pointer of the derived class
        ptr =   (derived *) &obj;            //correctreference
        //Other Program statements
}
```

Student Activity

1. Define Pointers.
2. What are the various operators of pointer? Describe theirusage.
3. How will you declare a pointer inC++?

Virtual Functions

Virtual functions, one of advanced features of OOP is one that does not really exist but it« appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class use_detined_name{
private:
public:
virtual return_type function_name1(arguments);
virtual return_type function_name2(arguments);
virtual return_type function_name3(arguments);
----------
};
```

To make a member function virtual, 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 precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

For. example, the following declararion of the virtual function is valid.

```
class point {
intx;
inty;
public:
virtual int length ( );
virtual void display ( );
};
```

Remember that 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.

Forexample

```
class point {
intx;
inty ;
public:
virtual void display ( );
};
virtual void point: : display ( ) //error
{
Function Body
}
```

A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class point {
int x ;
int y ;
public:
virtual static int length ( ); //error
```

```
};
        int point: : length ( )
        {
        Function body
        }
```
A virtual function cannot have a constructor member function but it can have the destructor member function.
```
class point {
int x ;
int y ;
public:
virtual point (int xx, int yy) ; // constructors, error
void display ( ) ;
int length ( ) ;
};
```
A destructor member function does not take any argument and no return type can be specified for it not even void.
```
class point {
int x ;
int y ;
public:
virtual point (int xx, int yy) ; //invalid
void display ( );
intlength ( ) ;
```
It is 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 virtuall method in the base class.
```
class base {
int x,y ;
public:
virtual int sum (int xx, int yy ) ; //error
} ;
class derived: public base {
intz ;
public:
virtual float sum (int xx, int yy) ;
};
```
The above declarations of two virtual functions are invalid. Even though these functions take identical arguments note that the return data types aredifferent.
```
virtual  int  sum  (int xx,intIT) ;//baseclass
virtual float sum (int xx, int IT) ; //derivedclass
```
Both the above functions can be written with int data types in the base class as well as in the derived class as
```
virtual intsum (int xx,int yy) ;//base class virtual
intsum (int xx, int yy) ;//derived class
```
Only a member function of a class can be declared as virtual. A non member function (nonmethod) of a class cannot be declaredvirtual.
```
virtual void display ( ) //error, nonmember function
{
        Function body
}
```

Late Binding

As we studied in the earlier unit, late binding means selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility. The late binding is implemented through virtual functions as a result we have to declare an object of a class either as a pointer to a class or a reference to aclass.

For example the following shows how a late binding or run time binding can be carried out with the help of a virtual function.

```
class base {
private :
int x;
float y;
public:
virtual void display ( ) ;
int sum ( ) ;
};
class derivedD : public baseA
{
private :
int x ;
float y;
public:
void display ( ); //virtual
int sum ( ) ;
};
void main ( )
{
        baseA *ptr ;
        derivedD objd ;
        ptr = &objd ;
        Other Program statements
        ptr- >di splay ( ) ; //run time binding ptr-
        >sum ( ) ; //compile time binding
}
```

Note that the keyword virtual is be followed by the return type of a member function if a run time is to be bound. Otherwise, the compile time binding will be effected as usual. In the above program segment, only the display ( ) function has been declared as virtual in the base class, whereas the sum ( ) is nonvirtual. Even though the message is given from the pointer of the base class to the objects of the derived class, it willnot

access the sum ( ) function of the derived class as it has been declared as nonvirtual. The sum ( ) function compiles only the static binding.

The following program demonstrates the run time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As function are declared as virtual, the C++ compiler invokes the dynamic binding.

```
#include <iostream.h>
#include <conio.h>
class baseA {
public :
virtual void display () {
cout<< “One \n”;
}
};
class derivedB : public baseA
{
        public:
        virtual void display(){
        cout<< “Two\n”; }
};
class derivedC: public derivedB
{
        public:
        virtual void display ( ) {
        cout<< “Three \n”; }
};
void main ( ) {
        //define three objects
        baseA obja;
        derivedB objb;
        derivedC objc;
        base A *ptr [3]; //define  an array of pointers to baseA
        ptr [0] =&obja;
        ptr [1] =  &objb;
        ptr [2] =&objc;
        for ( inti = 0; i <=2; i ++ )
        ptr [i]->display ( ); //same message for all objects
        getche ( );
}
Output
One
Two
Three
```

The program listed below illustrates the static binding of the member functions of a class. In program there are two classes student and academic. The class academic is derived from class student. The two member function getdata and display are defined for both the classes. *obj is defined for class student, the address of which is stored in the object of the class academic. The functions getdata ( ) and display ( ) of student class are invoked by the pointer to theclass.

```
#include<iostream.h>
#include<conio.h>
class student {
private:
int rollno;
char name [20];
public:
void getdata ( );
void display ( );
```

```
};
class academic: public student {
private:
char stream;
public:
void getdata ( );
void display ( ) ;
};
void student:: getdata ( )
{
        cout<< "enterrollno\n";
        cin>>rollno;
        cout<< "enter name \n";
        cin>>name;
}
void student:: display ( )
{
cout<< "the student's roll number is "<<rollno<< "and name is"<<name ;
cout<< endl;
}
void academic :: getdata ( )
{
        cout<< "enter stream of a student? \n";
        cin >>stream;
}
void academic :: display ( ) {
        cout<< "students stream \n";
        cout <<stream<<endl;
}
void main ( )
{
        student  *ptr  ;
        academic obj ;
        ptr=&obj;
        ptr->getdata ( ) ;
        ptr->display ( ) ;
        getche ( );
}
output
enter rollno
25
enter name
raghu
the student's roll number is 25 and name is raghu
```

The program listed below illustrates the dynamic binding of member functions of a class. In this program there are two classes student and academic. The class academic is derived from student. Student function has two virtual functions getdata ( ) and display (). The pointer for student class is defined and object . for academic class is created. The pointer is assigned the address of the object and function of derived class are invoked by pointer to student.

```
#include <iostream.h>
#include <conio.h>
class student {
```

```
private:
introllno;
char name [20];
public:
virtual void getdata ( );
virtual void display ( );
};
class academic: public student {
private :
char stream[10];
public:
void getdata { };
void display ( ) ;
};
void student: : getdata ( )
{
        cout<< “enter rollno\n”;
        cin >> rollno;
        cout<< “enter name \n”;
        cin >>name;
}
void student:: display ( )
{
cout<< “the student’s roll number is”<<rollno<< “and name is”<<name;
cout<< end1;
}
void academic: : getdata ( )
{
        cout << “enter stream of a student? \n”;
        cin>> stream;
}
void academic:: display ( )
{
        cout<< “students stream \n”;
        cout<< stream << endl;
}
void main ( )
{
        student *ptr ;
        academic obj ;
        ptr = &obj ;
        ptr->getdata ( );
        ptr->dlsplay ( );
        getch ( );
}
output
enter stream of a student?
Btech
students stream
Btech
```

Pure Virtual Functions

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task.

The following program demonstrates how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class. In the example there are two classes employee and grade. The class employee is base class and the grade is derived class. The functions getdata ( ) and display ( ) are declared for both the classes. For the class employee the functions are defined with empty body or no code inside the function. The code is written for the grade class. The methods of the derived class are invoked by the pointer to the base class.

```
#include<iostream.h>
#include<conio.h>
class employee {
int code
char name [20] ;
public:
virtual void getdata ( ) ;
virtual void display ( ) ;
};
class grade: public employee
{
        char grd [90] ;
        float salary ;
public :
        void getdata ( ) ;
        void display ( );
};
void employee :: getdata ( )
{
}
void employee:: display ( )
{
}
void grade : : getdata ( )
{
        cout<< " enter employee's grade ";
        cin>> grd ;
        cout<< "\n enter the salary " ;
        cin>> salary;
}
void grade : : display ( )
{
        cout« " Grade salary \n";
        cout« grd« ""« salary« endl;
```

```
}
void main ( )
{
```

```
        }
Output
employee *ptr ; grade obj ;
ptr = &obj ;
ptr->getdata ( ) ; ptr->display (
) ; getche ( ) ;
```

```
enter employee's grade A
enter the salary 250000
Grade salary
    A       250000
```

Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtualfunction.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can useits interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstracttoo.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

virtual void f() = 0;

Example of Abstract Class

```
classBase          //Abstract baseclass

public:
 virtual void show()=0;            //Pure VirtualFunction
};

class Derived:public Base
{
 public:
 void show()
 { cout <<"Implementation of Virtual Function in Derived class"; }
```

```
};

int main()
{
 Baseobj;       //Compile TimeError
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output : Implementation of Virtual Function in Derived class

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
// Base class
class Animal {
  public:
    void animalSound() {
    cout << "The animal makes a sound \n" ;
  }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
    cout << "The pig says: wee wee \n" ;
  }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
    cout << "The dog says: bow wow \n" ;
  }
};
```

UNIT-IV

Managing Console I/O

One of the most essential features of interactive programming is its ability to interact with the users through operator console usually comprising keyboard and monitor. Accordingly, every computer language (and compiler) provides standard input/output functions and/or methods to facilitate console operations.

C++ accomplishes input/output operations using concept of stream. A stream is a series of bytes whose value depends on the variable in which it is stored. This way, C++ is able to treat all the input and output operations in a uniform manner. Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply astream.

We have used the objects cin and cout (pre-defined in the iostream.h file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >>operator is overloaded in the istream class and « is overloaded in the ostream class.The following is the general format for reading data from the keyboard:

cin >> variable1 >> variable2 >>… …>> variableN;

Where variable1, variable2, are valid C++ variable names that have been declared already. This statement will cause the computer to halt the execution and look for input data from the keyboard. The input data for this statement would be:

data1 data2. dataN

The input data are separated by white spaces and should match the type of variable in the cin list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destinationtype.

For example, consider the following code:

```
int code;
cin >> code;
```

Suppose the following data is given as input:

1267E

The operator will read the characters up to 7 and the value 1267 is assigned to code. The character E remains in the input stream and will be input to the next cin statement. The general format of outputtingdata:

cout << iteml <<item2 <<<< itemN;

The items, item1 through itemN may be variables or constants of any basic types.

The put() and get() Functions

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object. For instance, look at the code snippet given below:

```
char c;
cin.get (c); //get a character from keyboard and assign it to c
while (c!= '\n')
{
        cout<<C;        //display the character on screen cin.get(c);
                        //get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator>>can also be used to read a character but it will skip the white spaces and newline character. The above while loop will not work properly if the statement

```
cin >> c;
```

is used in place of

```
cin.get (c);
```

Try using both of them and compare the results. The get(void) version is used as follows:

```
char c;
c-cin.getl);      //cin.get (c)replaced
```

The value returned by the function get() is assigned to the variable c.

The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

cout << put ('x');
displays the character xand
cout << put (ch);
displays the value of variable ch.

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

cout << put (68) ;

displays the character D. This statement will convert the int value 90 to a char value and display the character whose ASCII value is 68,

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;.
cin.get(c)        //read a character
while (c!='\n')
{
     cout<< put(c); //display the character on screen cin.get (c ) ;
}
```

The getline () and write () Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

cin.getline(line, size);

This function call invokes the function which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size number of characters are read (whichever occurs first). The newline. character is read but not saved. Instead, it is replaced by the nullcharacter.

For example; consider the following code:

```
char name [20] ;
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

Neeraj good

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

Object Oriented Programming

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

After reading the string/ cin automatically adds the terminating null character to thecharacter array.

Remember,thetwoblankspacescontainedinthestringarealsotakenintoaccount,i.e. between Objects and Oriented and Pro.

We can also read strings using the operator >>as follows:

cin >> name;

But remember cin can read strings that do not contain white space. This means that cin can read just one word and not a series of words such as “Neeraj good”.

Formatted Console I/O Operations

C++ supports a number of features that could be used for formattingtheoutput. These featuresinclude:

- ios class functions andflags.
- Manipulators.
- User-defined outputfunctions.

The ios class contains a large number of member functions that could be used to format the output in a number of ways. The most important ones among them are listed below.

Table 10.1

Function	Task
width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field.
self()	To specify format flags that can control the form of output display (such as Left-justification and right-justification).
Unself()	To clear the flags specified.

Manipulators are special functions that can be included in these statements to alter the format parameters of a stream. The table given below shows some important! manipulator functions that are frequently used. To access these manipulators, the file iomanip.h should be included in the program.

Table 10.2

Manipulator	Equivalent Ios function
setw()	width()
Setprecision()	Precision()
Setfill()	fill()
setiosflags()	self()
Resetiosflags()	Unself()

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats.

Streams

C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the system supplies an interface to the programmer that is independent of the actual device being accessed, This interface is known as stream.

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the output stream. In other words, a program extracts the bytes from an input stream and inserts bytes into an outputstream.

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device. Therefore, a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. They include cin and cout which have been used very often in our earlier programs. We know that cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, ifnecessary.

I/O Operations

Input and Output statements of computer languages are used to provide commu-nications between the user and the program. In most of the computer languages, input and output are done

through statements. But in C++, these operations are carried out through its built-in functions. The I/O functions are designed in header files like fstream.h, iostream.h etc.

Through these functions, data can be read from or written to files or standard input/output devices like keyboard and VDU. This execution of a program can be interrupted by input/output calls. Hence the data can be entered or output can be retrieved during execution.

The file, stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, pot( ) and get( ), are designed for handling a single character at a time. Another pair of functions,

write() and readQ, are designed to write and read blocks of binary data.

put( ) and get( )Functions

The function put( ) writes a single character to the associated stream. Similarly, th**e** function get ( ) reads a single character from the, associated stream. The program, requests for a string. On receiving the string, the program writes it, character, by character, to the file using the pot() function in a for loop. Note that the length of the string is used to terminate the for loop.

C++ provides a number of useful predefined stream classes for console input/output operations. Some of the C++ the predefined stream objects are listed below.

cin This is the name ofstandard input stream, usually keyboard. The corresponding name in C isstdin.

cout This is the name of standard output stream, usually screen of the monitor. The corresponding name in C isstdout.

cerr This is the name of standard error output stream, usually screen of the monitor. The corresponding name in C isstderr.

clog This is another version of cerr. It provides buffer to collect errors. C does not have a stream equivalent tothis.

In their default roles, these streams are tied up with the keyboard and screen of the monitor as describe above. However, you can redirect them from and to other devices and files.

Working With Files

Streams: C++ is designed to work with a wide variety of devices including, disks and take drives. Although each device is very different the system suppliers an interface to the programmer that is independent of the actual device accessed. This interface is known as stream.

1. Ofstream: This file handling class in C++ signifies the output file stream and is applied to create files for writing information to files
2. Ifstream: This file handling class in C++ signifies the input file stream and is applied for reading information from files
3. Fstream: This file handling class in C++ signifies the file stream generally, and has the capabilities for representing both ofstream and ifstream

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

Opening and Closing a File in C++

If programmers want to use a disk file for storing data, they need to decide about the following things about the file and its intended use. These points that are to be noted are:

- A name for the file
- Data type and structure of the file
- Purpose (reading, writing data)
- Opening method
- Closing the file (after use)

Files can be opened in two ways. They are:

1. Using constructor function of the class
2. Using member function open of the class

Opening a File

The first operation generally performed on an object of one of these classes to use a file is the procedure known as to opening a file. An open file is represented within a program by a stream and any input or output task performed on this stream will be applied to the physical file associated with it. The syntax of opening a file in C++ is:

open (filename, mode);

There are some mode flags used for file opening. These are:

- ios::app: append mode
- ios::ate: open a file in this mode for output and read/write controlling to the end of the file
- ios::in: open file in this mode for reading
- ios::out: open file in this mode for writing
- ios::trunk: when any file already exists, its contents will be truncated before file opening

Closing a file in C++

When any C++ program terminates, it automatically flushes out all the streams releases all the allocated memory and closes all the opened files. But it is good to use the close() function to close the file related streams and it is a member of ifsream, ofstream and fstream objects.

The structure of using this function is:

void close();

General functions used for File handling

1. open(): To create a file
2. close(): To close an existing file
3. get(): to read a single character from the file
4. put(): to write a single character in the file
5. read(): to read data from a file
6. write(): to write data into a file

Reading from and writing to a File

While doing C++ program, programmers write information to a file from the program using the stream insertion operator (<<) and reads information using the stream extraction operator (>>). The only difference is that for files programmers need to use an ofstream or fstream object instead of the cout object and ifstream or fstream object instead of the cin object.

Example:

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

void main () {
  ofstream file;
  file.open ("egone.txt");
  file << "Writing to a file in C++....";
  file.close();
  getch();
}
```

Another Program for File Handling in C++

Example:

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

void main()
{
  char c,fn[10];
  cout<<"Enter the file name....:";
  cin>>fn;
  ifstream in(fn);
  if(!in)
```

```
 {
  cout<<"Error! File Does not Exist";
  getch();
  return;
 }
  cout<<endl<<endl;
  while(in.eof()==0)
  {
    in.get(c);
    cout<<c;
  }
  getch();
}
```

Another C++ Program to Print Hello GS to the Console

Example:

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

void main()
{
 ofstream fileo("Filethree");
 fileo<<"Hello GS";
 fileo.close();
 ifstream fin("Filethree");
 char ch;
 while(fin)
 {
  fin.get(ch);
  cout<<ch;
 }
 fin.close();
 getch();
}
```

Template:

Template supports generic programming, which allows developing reusable software components such as functions, classes, etc supporting different data types in a single frame work.

A template in c++ allows the construction of a family of template functions and classes to perform the same operation o different data types. The templates declared for functions are called class templates. They perform appropriate operations depending on the data type of the parameters passed tothem.

Function Templates:

A function template specifies how an individual function can be constructed.

template <class T>
return type functionnm(T arg1,T arg2)
{
fn body;
}

For example:
Input two number and swap their values

template <class T>
void swap (T &x,T & y)

```
{
T  z;
z=x;
x=y;
y=z;
}
void main( )
{
char ch1,ch2;
cout<<”enter two characters:”;
cin>>ch1>>ch2;
swap(ch1,ch2);
cout<<ch1<<ch2;
int a,b;
cout<<”enter a,b:”;
cin>>a>>b;
swap(a,b);
cout<<a<<b;
float p,q;
cout<<”enter p,q:”;
cin>>p>>q;
swap(p,q);
cout<<p<<q;
}
```

example 2:
find maxium between two data items.

```
template <class T>
T max(T a,T b)
```

```
{
if (a>b)
return a;
else
return b;
}
void main()
{
char ch1,ch2;
cout<<"enter two characters:";
cin>>ch1>>ch2;
cout<<max(ch1,ch2);
int a,b;
cout<<"enter a,b:";
cin>>a>>b;
cout<<max(a,b);
float p,q;
cout<<"enter p,q:";
cin>>p>>q;
cout<<max(p,q);
}
```

Overloading of function template

```
#include<iostream.h>
template <class T>
void print( T a)
{
        cout<<a;
}
template <class T>
void print( T a, int n)
{
int i;
for(i=0;i<n;i++)
        cout<<a;
}
void main()
{
print(1);
print(3.4);
print(455,3);
print("hello",3);
}
```

Multiple arguments function template:

find sum of two different numbers

```
template <class T,class U>
T sum(T a,U b)
{
        return a+(U)b;
}
void main( )
```

```
{
cout<<sum(4,5.5);
cout<sum(5.4,3);
}
```

Class Template

similar to functions, classes can also be declared to operate on different data types. Such classes are class templates. a class template specifies how individual classes can be constructed similar to normal class definition. These classes model a generic class which support similar operations for different datatypes.

syn:

```
template <class T>
class classnm
{
T member1;
T member2;
…
…
public:
T fun();
…
..
};
```

objects for class template is created like:

```
classnm <datatype> obj;
obj.memberfun();
```

Exception Handling:

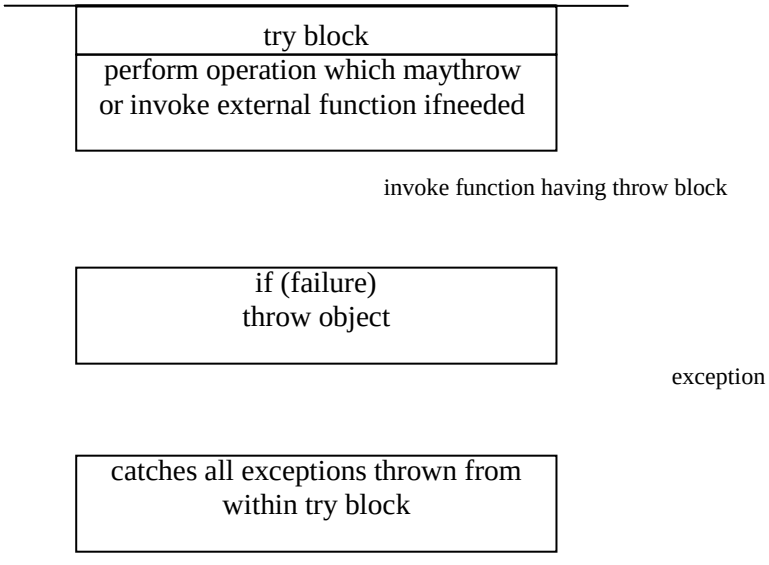
Exception refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error handling mechanism of c++ is generally referred to as exception handling.

Generally , exceptions are classified into synchronous and asynchronous exceptions.. The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data. with in a program is known as synchronous exception.
Example:
errors such as out of range,overflow,underflow and so on.

The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.
For example, errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

exception handling model:

When a program encounters an abnormal situation for which it in not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception. The exception handling mechanism uses three blocks: try, throw and catch.
The try block must be followed immediately by a handler, which is a catch block. If an exception is thrown in the try block the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. The relationship of these three exceptions handling constructs called the exception handling model is shown in figure:



throw construct:

The keyword throw is used to raise an exception when an error is generated in the comutation. the throw expression initialize a temporary object of the typeT used in thorw (T arg).
syntax:

```
        throw T;
```

catch construct:

The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword. The catch handler can also occur immediately after another catch Each handler will only evaluate an exception that matches.
syn:

```
catch(T)
{
// error meassges
}
```

try construct:
The try keyboard defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted.

```
try
{
 …
if (failure)
        throw T;
}
catch(T)
{
…
}
```

example:

```
#include<iostream.h>
void main()
{
int a,b;
cout<<"enter two numbers:";
cin>>a>>b;
try
{
if (b= =0)
        throw b;
else
        cout<a/b;
}
catch(int x)
{
cout<<"2nd operand can't be 0";
}
}
```

Array reference out of bound:

```
#define max 5
class array
{
private:
        int a[max];
public:
        int &operator[](int i)
        {
        if (i<0 || i>=max)
                throw i;

        else
                return a[i];
        }
};
void main()
{
array x;
try
{
cout<<"trying to refer a[1]…"
x[1]=3;
cout<<"trying to refer a[13]…"
x[13]=5;
}
catch(int i)
{
cout<<"out of range in array references…";
}
}
```

multiple catches in a program

```
void test(int x)
{
try{
if (x==1)
        throw x;
else if (x==-1)
        throw 3.4;
else if (x==0)
        throw 's';
}
catch (int i)
{
cout<<"caught an integer…";
}
catch (float s)
{
cout<<"caught a float…";
```

```
}
catch (char c)
{
cout<<"caught a character…";
}}
void main()
{
test(1);
test(-1);
test(0);
}
```

catch all

```
void test(int x)
{
try{
if (x==1)
        throw x;
else if (x==-1)
        throw 3.4;
else if (x==0)
        throw 's';
}
catch (…)
{
cout<<"caught an error…";
}
```

Unit:V

Standard Template Library:

The C++ Standard Library can be categorized into two parts −

- **The Standard Function Library** − This library consists of general-purpose,stand-alone functions that are not part of any class. The function library is inherited from C.
- **The Object Oriented Class Library** − This is a collection of classes and associated functions.

Standard C++ Library incorporates all the Standard C libraries also, with small additions and changes to support type safety.

The Standard Function Library

The standard function library is divided into the following categories −

- I/O,
- String and character handling,
- Mathematical,
- Time, date, and localization,

- Dynamic allocation,
- Miscellaneous,
- Wide-character functions,

The Object Oriented Class Library

Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following −

- The Standard C++ I/O Classes
- The String Class
- The Numeric Classes
- The STL Container Classes
- The STL Algorithms
- The STL Function Objects
- The STL Iterators
- The STL Allocators
- The Localization library
- Exception Handling Classes
- Miscellaneous Support Library

Manipulating Strings:

A string is a sequence of character. As you know that C++ does not support built-in string type, you have used earlier those null character based terminated array of characters to store and manipulate strings. These strings are termed as *C Strings*. It often becomes inefficient performing operations on C strings. Programmers can also define their own string classes with appropriate member functions to manipulate strings. ANSI standard C++ introduces a new class called string which is an improvised version of C strings in several ways. In many cases, the strings object may be treated like any other built-in data type. The string is treated as another container class for C++.

Table of Contents

The C Style String

The C style string belongs to C language and continues to support in C++ also strings in C are the one-dimensional array of characters which gets terminated by \0 (null character).

This is how the strings in C are declared:
char ch[6] = {'H', 'e', 'l', 'l', 'o', '

char ch[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

'};
Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the \0 at the end of the string when it initializes the array.

String Class in C++

The string class is huge and includes many constructors, member functions, and operators.
Programmers may use the constructors, operators and member functions to achieve the following:

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string
- Adding objects of string
- Comparing strings
- Accessing characters of a string
- Obtaining the size or length of a string, etc...

Manipulate Null-terminated strings

C++ supports a wide range of functions that manipulate null-terminated strings. These are:

- strcpy(str1, str2): Copies string str2 into string str1.
- strcat(str1, str2): Concatenates string str2 onto the end of string str1.
- strlen(str1): Returns the length of string str1.
- strcmp(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- strchr(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- strstr(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- append(): This function appends a part of a string to another string
- assign():This function assigns a partial string
- at(): This function obtains the character stored at a specified location
- begin(): This function returns a reference to the start of the string

- capacity(): This function gives the total element that can be stored
- compare(): This function compares a string against the invoking string
- empty(): This function returns true if the string is empty
- end(): This function returns a reference to the end of the string
- erase(): This function removes character as specified
- find(): This function searches for the occurrence of a specified substring
- length(): It gives the size of a string or the number of elements of a string
- swap(): This function swaps the given string with the invoking one

Important Constructors obtained by String Class

- String(): This constructor is used for creating an empty string
- String(const char *str): This constructor is used for creating string objects from a null-terminated string
- String(const string *str): This constructor is used for creating a string object from another string object

Operators used for String Objects

1. =: assignment
2. +: concatenation
3. ==: Equality
4. !=: Inequality
5. <: Less than
6. <=: Less than or equal
7. >: Greater than
8. >=: Greater than or equal
9. []: Subscription
10. <<: Output
11. >>: Input

Object Oriented Systems Development:

The software development process consists of
(1) Analysis – Translates the user needs into system requirements and responsibilities.
(2) Design – It begins with a problem statement and ends with a detailed design that can be transformed into an operational system.
(3) Implementation – It regines the detailed design into the system deployment that will satisfy the users needs.
(4) Testing – Two basic approaches to system testing, they are (i) Test according to how it has been built for. (ii) What it should do?
(5) Maintenance.

Object Oriented Systems Development Life Cycle (SDLC)

This is also known as Classic Life Cycle Model (or) Linear Sequential Model (or) Waterfall Method. This model has the following activities.

- System/Information Engineering and Modeling

As software is always of a large system (or business), work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must interface with other elements such as hardware, people and other resources. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, the system should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

- Software Requirement Analysis

This process is also known as feasibility study. In this phase, the development team visits the customer and studies their system. They investigate the need for possible software automation in the given system. By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc.. The requirement gathering process is intensified and focused specially on software. To understand the nature of the program(s) to be built, the system engineer or "Analyst" must understand the information domain for the software, as well as required function, behavior, performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved.

- System Analysis and Design

In this phase, the software development process, the software's overall structure and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc.. are all defined in this phase. A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

- Code Generation

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like compilers, interpreters, debuggers etc.. are used to generate the code. Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

- Testing

Once the code is generated, the software program testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations.

- Maintenance

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

Prototyping Model

This is a cyclic version of the linear model. In this model, once the requirements analysis is done and the design for a prototype is made, the development process gets started. Once the prototype is created, it is given to the customer for evaluation. The customer tests the package and gives his/her feed back to the developer who refines the product according the customer's exact expectation. After a finite number of iterations, the final software package is given to the customer. In this methodology, the software is evolved as a result of periodic shuttling of information between the customer and developer. This is the most popular development model in the contemporary IT industry. Most of the successful software products have been developed using this model – as it is very difficult to comprehend all the requirements of a customer in one shot. There are many variations of this model skewed with respect to the project management styles of the companies. New versions of a software product evolve as a result of prototyping.

Rapid Application Development (RAD) model

The RAD models a linear sequential software development process that emphasizes an extremely short development cycle. The RAD model is a "high speed" adaptation of the linear sequential model in which rapid development is achieved by using a component-based construction approach. Used primarily for information systems applications, the RAD approach encompasses the following phases,

- **Business modeling,**
- **Data modeling,**
- **Process modeling,**
- **Application generation,**
- **Testing and turnover**.

Component Assembly Model

Object technologies provide the technical framework for a component based process model for software engineering. The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithm that are used to manipulate the data. If properly designed and implemented, object oriented classes are reusable across different applications and computer based system architectures. Component Assembly Model leads to software reusability. The integration/assembly of the already existing software components accelerates the development process. Nowadays many component libraries are available on the Internet. If the right components are chosen, the integration aspect is made much simpler.

****