UNIT I

Syllabus: Introduction to C++: Difference between C and C++, Evolution of C++, The Object Oriented Technology, Disadvantage of Conventional Programming, Key Concepts of Object Oriented Programming, Advantage of OOP, Object Oriented Language.

1. **Introduction:**

C++ is an object oriented programming language, C++ was developed by Jarney Stroustrup in 1983 at AT & T Bell laboratories, USA. C++ was developed from C and simula 67 language. C++ was early called 'C with classes'.

- C++ is derived from C Language. It is a Superset of C.
- Earlier C++ was known as C with classes.
- In C++, the major change was the addition of classes and a mechanism for inheriting class objects into other classes.
- Most C Programs can be compiled in C++ compiler.
- C++ expressions are the same as C expressions.
- All C operators are valid in C++.

2. Differences between C and C++

Following are the differences Between C and C++ :

S.No.	C	C++
1	C is Procedural Language.	C++ is non Procedural i.e Object oriented Language.
2	No virtual Functions are present in C	The concept of virtual Functions are used in C++.
3	In C, Polymorphism is not possible.	The concept of polymorphism is used in C++. Polymorphism is the most Important Feature of OOPS.

4	Operator overloading is not possible in C.	Operator overloading is one of the greatest Feature of C++.
5	Top down approach is used in Program Design.	Bottom up approach adopted in Program Design.
6	No namespace Feature is present in C Language.	Namespace Feature is present in C++ for avoiding Name collision.
7	Multiple Declaration of global variables are allowed.	Multiple Declaration of global varioables are not allowed.
8	In C • scanf() Function used for Input. • printf() Function used for output.	In C++ • Cin>> Function used for Input. • Cout<< Function used for output.
9	Mapping between Data and Function is difficult and complicated.	Mapping between Data and Function can be used using "Objects"
10	In C, we can call main() Function through other Functions	In C++, we cannot call main() Function through other functions.
11	C requires all the variables to be defined at the starting of a scope.	C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12	No inheritance is possible in C.	Inheritance is possible in C++
13	In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	In C++, **new** and **delete** operators are used for Memory Allocating and Deallocating.
14	It supports built-in and primitive data	It support both built-in and user

	types.	define data types.
15	In C, does not provide direct support to Eexception handling.	In C++, Exception Handling is done with Try and Catch block.

3. **Evolution of C++**

C++ is an object oriented programming language and also considered as an extension of C. Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey (USA) developed this language in the early 1980s. Stroustrup, a master of Simula67 and C, wanted to combine the features of both the languages and he developed a powerful language that supports object-oriented programming with features of C. The outcome was C++ as per Fig. 1.1. Various features were derived from ***SIMULA67*** and ***ALGOL68***. Stroustrup called the new language ***'C with classes'***. However, in 1983, the name was changed to C++.

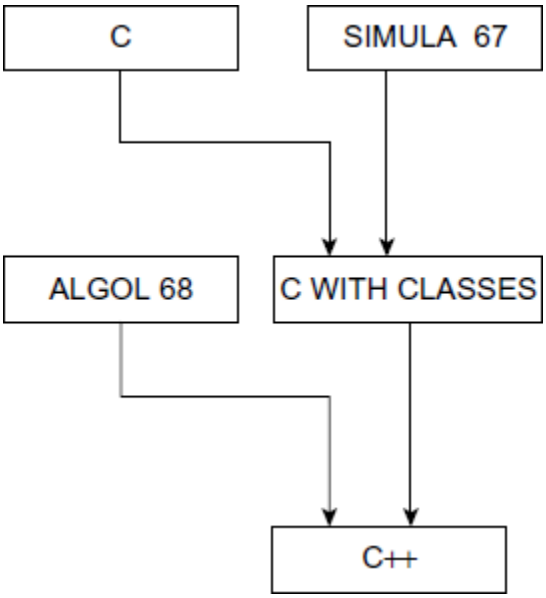


Fig. 1.1 Evolution of C++

The thought of C++ came from the C increment operator ++. Rick Mascitti coined the term C++ in 1983. Therefore, C++ is an extension of C. C++ is a superset of C. All the concepts of C are applicable to C++ also.

For developing complicated applications, object oriented language such as C++ is the most convenient and easy. Hence, a programmer must be aware of its features.

4. THE OBJECT ORIENTED TECHNOLOGY

Nature is composed of various objects. Living beings can be categorized into different objects.

Let us consider an example of a teaching institute which has two different working sections – teaching and non-teaching. Further sub-grouping of teaching and non-teaching can be made for the coordination of management. The various departments of any organization can be thought of as objects working for certain goals and objectives.

Usually an institute has faculty of different departments. The Director/Principal is a must for the overall management of the institute. The Academic Dean is responsible for the academics of the institute. The Dean for Planning should have the future plans of the institute and he/she must see how the infrastructure is utilized effectively. The Dean R&D should see research activities run in the institute forever.

Besides teaching staff there must be laboratory staff for assistance in conducting practical sessions, and a site development section for beautification of the campus. The accounts department is also required for handling monetary transactions and salaries of the employees. The Sports section is entrusted the responsibility of sports activities. The Registrar for Administration and staff for dealing with administrative matters of the institute are also required. Each department has an in-charge who carries clear-cut given responsibilities. Every department has its own work as stated above. When an institute's work is distributed into departments as shown in Fig. 1.2, it is comfortable to accomplish goals and objectives. The activities are carried on smoothly. The burden of one particular department has to be shared among different departments with personnel. The staff in the department is controlled properly and act according to the instructions laid down by the management. The faculty performs activities related to teaching. If the higher authority needs to know the details regarding the theory, practical, seminar and project loads of individuals of the department, then a person from the department furnishes the same to the higher authority. This way some responsible person from the department accesses the data and provides the higher authority with the requisite information. It is also good to think that no unconnected person from

another department reads the data or attempts to make any alteration that might corrupt the data.

As shown in Fig. 1.2, an institute is divided into different departments such as library, classroom, computer laboratory, etc. Each department performs its own activities in association with the other departments. Each department may be considered as a module and it contains class and object in C++ language. This theory of class and object can be extended to every walk of life and can be implemented with software. In general, objects are in terms of entities.

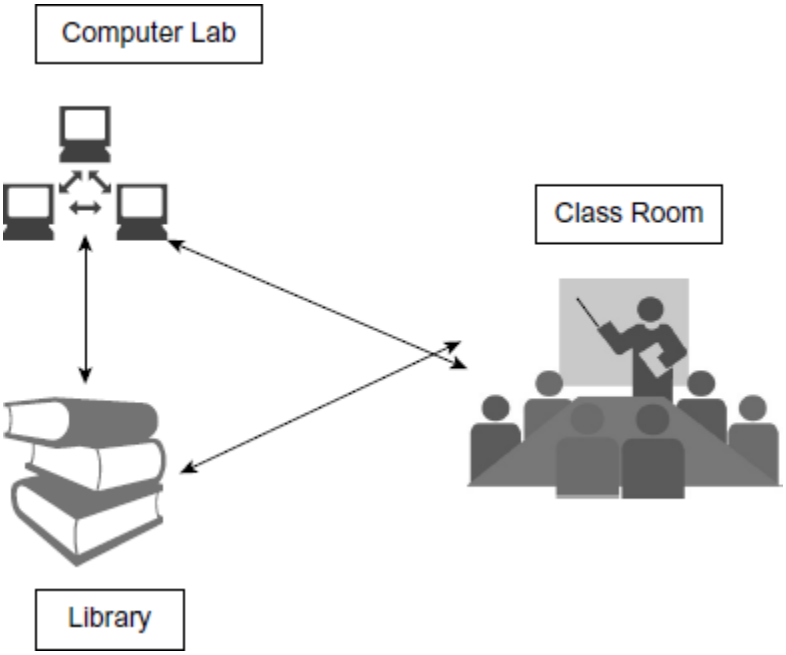


Fig.1.2 Relationship between different sections

In a nutshell, in object oriented programming objects of a program interact by sending messages to each other.

5. DISADVANTAGE OF CONVENTIONAL PROGRAMMING

Traditional programming languages such as COBOL, FORTRAN, C etc. are commonly known as procedure oriented languages. The program written in these languages consists of a sequence of instructions that tells the compiler or interpreter to perform a given task. Numerous functions are initiated by the user to perform a task. When a program code is large, it becomes inconvenient to manage it. To overcome this problem, procedures or subroutines were adopted to make a

program more understandable to the programmers. A program is divided into many functions.

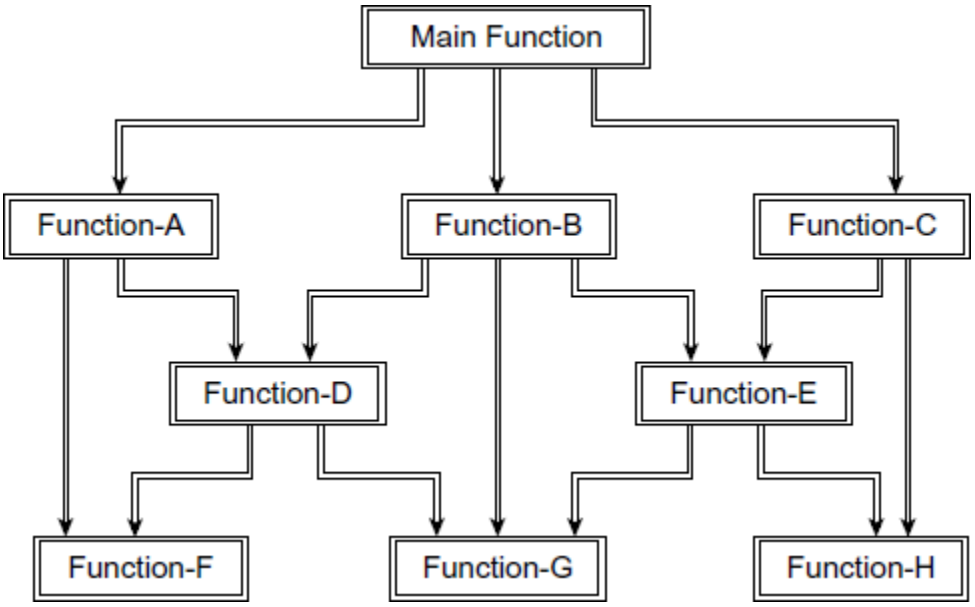


Fig. 1.3 Flow of functions in non-OOP languages

Each function can call another function, as shown in Fig. 1.3. Each function has its own task. If the program is too large the function also creates problems. In many programs, important data variables are declared as global. In case of programs containing several functions, every function can access the global data as per the simulation in Fig. 1.4. In huge programs it is difficult to know what data is used by which function. Due to this the program may contain several logical errors.

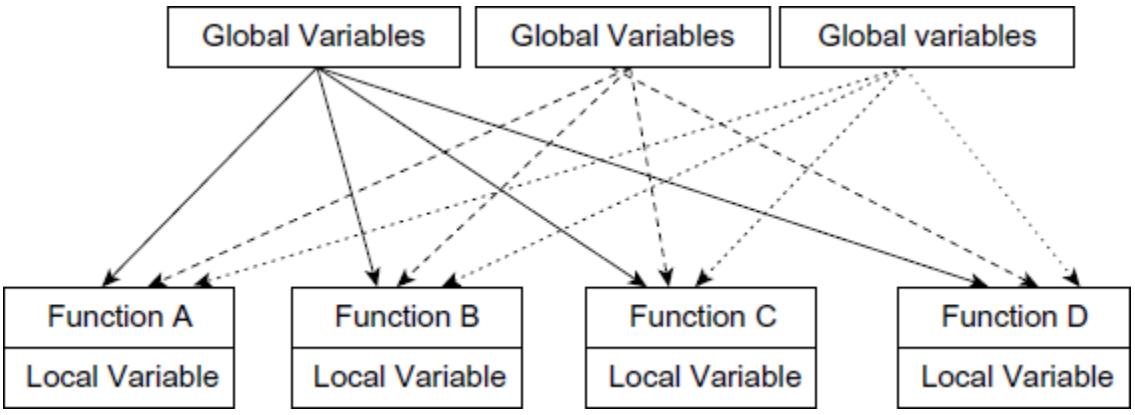


Fig. 1.4 Sharing of data by functions in non-OOP languages

The following are the drawbacks observed in monolithic, procedure, and structured programming languages:

1. Huge programs are divided into smaller programs known as functions. These functions can call one another. Hence security is not provided.
2. No importance is given to security of data and importance is laid on doing things.
3. Data passes globally from function to function.
4. Most function accesses global data.

6.Key Concepts of Object Oriented Programming

The major Concepts of Object Oriented Programming are:

1. Class
2. Object
3. Abstraction
4. Encapsulation
5. Data Hiding
6. Inheritance
7. Reusability
8. Polymorphism
9. Virtual Functions
10. Message passing

Class: Class is an abstract data type (user defined data type) that contains member variables and member functions that operate on data. It starts with the keyword class. A class denotes a group of similar objects.

```
e.g.:  class employee
       {
        int empno;
        char name[25],desg[25];
        float sal;
       public:
```

```
void getdata ();

void putdata ();

};
```

Object: An object is an instance of a class. It is a variable that represents data as well as functions required for operating on the data. They interact with private data and functions through public functions.

e.g.: employee e1, e2;

In the above example employee is the class name and e1 and e2 are objects of that class.

Abstraction: Abstraction refers to the process of concentrating on the most essential features and ignoring the details. There are two types of abstraction

i) Procedural Abstraction
ii) Data Abstraction

Procedural Abstraction: Procedural abstraction refers to the process of using user-defined functions or library functions to perform a certain task, without knowing the inner details. The function should be treated as a black box. The details of the body of the function are hidden from the user.

Data Abstraction: Data Abstraction refers to the process of formation of user defined data type from different predefined data types.

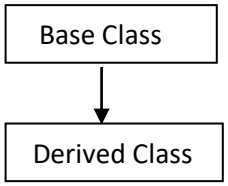
e.g: structure, class.

Encapsulation: Encapsulation is the process of combining data members and member functions into a single unit as a class in order to hide the internal operations of the class and to support abstraction.

Data Hiding: All the data in a class can be restricted from using it by giving some access levels (visibility modes). The three access levels are private, public, protected.

Private data and functions are available to the public functions only. They cannot be accessed by the other part of the program. This process of hiding private data and functions from the other part of the program is called as data hiding.

Inheritance: Inheritance is the process of acquiring (getting) the properties of some other class. The class whose properties are being inherited is called as base class and the class which is getting the properties is called as derived class.



Reusability: Using the already existing code is called as reusability. This is mostly used in inheritance. The already existing code is inherited to the new class. It saves a lot of time and effort. It also reduces the size of the program.

Polymorphism: Polymorphism means the ability to take many forms. Polymorphism allows to take different implementations for same name.

poly → many

morphism → forms

There are two types of polymorphism, Compile time polymorphism and run time polymorphism.

In Compile time polymorphism binding is done at compile time and in runtime polymorphism binding is done at runtime.

e.g.: Function overloading, operator overloading

Function Overloading: Function overloading is a part of polymorphism. Same function name having different implementations with different number and type of arguments.

Operator Overloading: Operator overloading is a part of polymorphism. Same operator can have different implementations with different data types.

Virtual Functions: Virtual functions are special type of functions which are defined in the base class and are redefined in the derived class. When virtual function is called with a base pointer and derived object then the derived class function will be called. A function can be defined as virtual by placing the keyword virtual for the member function.

Message Passing: An object-oriented program contains a set of objects that communicate with one another. The process of object oriented programming contains the basic steps:

1. Creating classes
2. Creating objects
3. Communication among objects

This communication is done with the help of functions (i.e., passing objects to functions)

7.Advantage of OOP

Object oriented technology provides many advantages to the programmer and the user. This technology solves many problems related to software development, provides improved quality and low cost software.

1. Object oriented programs can be comfortably upgraded.
2. Using inheritance, we can eliminate redundant program code and continue the use of previously defined classes.

3. The technology of data hiding facilitates the programmer to design and develop safe programs that do not disturb code in other parts of the program.
4. The encapsulation feature provided by OOP languages allows programmer to define the class with many functions and characteristics and only few functions are exposed to the user.
5. All object oriented programming languages allows creating extended and reusable parts of programs.
6. Object oriented programming changes the way of thinking of a programmer. This results in rapid development of new software in a short time.
7. Objects communicate with each other and pass messages.

Applications of OOPS

The promising areas for application of OOP includes:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and experttext
- AI and expertsystems
- Neural networks and parallel programming
- Decision support and Automation system
- CIM/CAM/CAD systems

8.Object Oriented Languages

There are many languages which support object oriented programming. Tables 1.1 and 1.2 describe the OOP languages and features supported by them.

Table 1.1 Properties of pure OOP and object based languages

	Pure Object Oriented Languages					Object Based Languages
Properties	**Java**	**Simula**	**Smalltalk**	**Eiffel**	**Java**	**Ada**
Encapsulation	✔	✔	✔	✔	✔	✔
Inheritance	✔	✔	✔	✔	✔	No
Multiple inheritance	✘	✘	✔	✔	✘	No
Polymorphism	✔	✔	✔	✔	✔	✔
Binding (Early and late)	Both	Both	Late binding	Early binding	Both	Early binding
Genericity	✘	✘	✘	✔	✘	✔
Class libraries	✔	✔	✔	✔	✔	Few
Garbage collection	✔	✔	✔	✔	✔	✘
Persistence	✔	✘	Promised	Less	✔	Same as 3GL
Concurrency	✔	✔	Less	Promised	✔	Hard

Table 1.2 Properties of extended traditional languages

	Extended traditional languages				
Properties	**Objective C**	**C++**	**Charm ++**	**Objective Pascal**	**Turbo Pascal**
Encapsulation	✓	✓	✓	✓	✓
Inheritance	✓	✓	✓	✓	✓
Multiple inheritance	✓	✓	✓	- - -	- - -
Polymorphism	✓	✓	✓	✓	✓
Binding (Early and late)	Both	Both	Both	Late	Early
Genericity	X	✓	✓	X	X
Class libraries	✓	✓	✓	✓	✓
Garbage collection	✓	X	X	✓	✓
Persistence	X	X	X	X	X
Concurrency	Poor	Poor	✓	X	X

The following are the object-oriented languages, which are widely accepted by the programmer.

- C++
- Smalltalk
- Charm ++
- Java

SMALLTALK

Smalltalk is a pure object oriented language. C++ makes few compromises to ensure quick performance and small code size. Smalltalk uses run-time binding. Smalltalk programs are considered to be faster than the C++. Smalltalk needs longer time to learn than C++. Smalltalk programs are written using Smalltalk browser. Smalltalk uses dynamic objects and memory is allocated from free store. It also provides automatic garbage collection and memory is released when object is no longer in use.

CHARM++

Charm ++ is also an object oriented programming language. It is a portable. The language provides features such as inheritance, strict type checking, overloading, and reusability. It is designed in order to work efficiently with different parallel systems together with shared memory systems, and networking.

Answer the following questions.

1. Compare C and C++.
2. What is object oriented programming?
3. Explain the key concepts of OOP.
4. What are the disadvantages of conventional programming languages?
5. Explain the characteristics of monolithic programming languages.
6. List the disadvantages of procedural programming languages.
7. Explain evolution of C++.
8. List the names of popular OOP languages.
9. List the unique features of an OOP paradigm.
10. What is an object and class?
11. How is data secured in OOP languages?
12. Compare and contrast OOP languages with procedure oriented languages.
13. Mention the types of relationships between two classes.
14. What is structured oriented programming? Discuss its pros and cons.
15. How is global data shared in procedural programming?
16. Describe any two object oriented programming languages.
17. What are the differences between Java and C++?
18. Mention the advantages of OOP languages?
19. What do you mean by message passing?
20. Distinguish between inheritance and delegation.

Answer the following by selecting the appropriate option.

1. Data hiding concept is supported by language
    1. C
    2. Basic
    3. Fortran
    4. **C++**
1. Function overloading means
    1. different functions with different names
    2. function names are same but same number of arguments
    3. **function names are same but different number of arguments**
    4. none of the above

3. C++ language was invented by
    1. Bjarne Stroupstrup
    2. Dennis Ritche
    3.Ken Thompson
    4.none of the above

4. The languages COBOL and BASIC are commonly known as
    1. procedure oriented languages
    2. object oriented languages
    3. low level languages
    4. none of the above

5. A program with only one function is observed in
    1. monolithic programming languages.
    2. object oriented languages
    3. structured programming languages
    4. none of the above

6.The packing of data and functions into a single component is known as
    1. encapsulation
    2. polymorphism
    3. abstraction
    4. none of the above

7. The method by which objects of one class get the properties of objects of another class is known as
    1. inheritance
    2. encapsulations
    3. abstraction
    4. none of the above

8.The mechanism that allows same functions to act differently on different classes is known as

1. **polymorphism**
2. encapsulations
3. inheritance
4. none of the above

9.The existing class can be reused by

1. **inheritance**
2. polymorphism
3. dynamic binding
4. abstraction

10.Composition of objects in a class is known as

1. **delegation**
2. inheritance
3. polymorphism
4. none of the above

11.A class

1. **binds the data and its related functions together**
2. data and their addresses
3. contains only functions
4. none of the above

12. The major drawback of procedural programming languages is

1. frequently invoking functions from `main()`
2. **non security of data**
3. non security of methods
4. none of the above

UNIT - II

Classes and Objects &Constructors and Destructor: Classes in C++, Declaring Objects, Access Specifiers and their Scope, Defining Member Function, Overloading Member Function, Nested class, Constructors and Destructors, Introduction, Constructors and Destructor, Characteristics of Constructor and Destructor, Application with Constructor, Constructor with Arguments parameterized Constructor, Destructors, Anonymous Objects.

What is a class? How can you define a class in C++? Explain with an example.

Object Oriented Programming encapsulates data (attributes) and functions (behavior) into packages called classes.

The class combines data and methods for manipulating that data into one package. An object is said to be an instance of a class. A class is a way to bind the data and its associated functions together. It allows the data to be hidden from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type.

Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The **class declaration** describes the type and scope of its members. The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

```
class class_name
{
        private:
                variable declarations;
                function declarations;
        public:
                variable declarations;
                function declarations;
};
```

- ✓ The class declaration is similar to structure declaration in C. The keyword **class** is used to declare a class. The body of a class is enclosed within braces and terminated by a semicolon.

- ✓ The class body contains the declaration of variables and functions. These functions and variables are collectively called members. They are usually grouped under two sections i,e. private and public

to denote which members are private and which are public. These keywords are known as **visibility labels**.

- ✓ The members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also.

- ✓ The data hiding is the key feature of OOP. By default, the members are **private.** The variables declared inside the class are known as data members and the functions are known as *member functions*.

- ✓ Only the member functions can have access to the private members and functions. However, the public members can be accessed from outside the class.

- ✓ The binding of data and functions together into a single class type variable is referred to as *encapsulation.*

A Simple Class Example: **A typical class declaration would look like: class item**

```
{
      int no;              // variables declaration float cost;
                           // private by default
   public:
      void getdata(int a, float b); // functions declaration void
      putdata(void);                // using prototype
};
```

How can you declare objects to a class? Explain with an example.

Once a class has been declared, we can create variables of that type by using the class name. For example,

item x; // memory for x is created Creates a variable x of type *item*.

In C++, the class variables are known as *objects.* Therefore, x is called an object of type item. We may also declare more than one object in one statement. Example:

item x, y, *z;

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage.

Note that class specification provides only a *template* and does not create any memory space for the objects.

How can you access the class members?
The object can access the public class members of a class by using dot operator or arrow operator. The syntax is as follows;

Objectname operator membername;

Example:

```
x.show();
z->show(); {z is a pointer to class item}
```

What is an access specifier? Explain about various access specifiers and their scope.
The access specifier specifies the accessibility of the data members declared inside the class. They are:

1. public
2. private
3. protected

public keyword can be used to allow object to access the member variables of class directly. The keyword **public** followed by colon (:) means to indicate the data member and member function that visible outside the class.
Consider the following example:

```
class Test
{
    public:
       int x, y;     // variables declaration
       void show()
       {
             cout<<x<<y;
       }
};

int main()
{
   Test t;          //Object creation
   t.x = 10;
   t.y = 20;
   t.show();
};
```

Now, it display 10 and 20

Private keyword is used to prevent direct access to member variables

or functions by the object. It is the **default access**. The keyword **private** followed by colon (:) is used to make data member and member function visible with in the class only.

Consider the following example:

```
class Test
{
    private:
       int x, y;    // variables declaration
};

int main()
{
   Test t;          //Object creation
   t.x = 10;
   t.y = 20;
};
```

Now it raises two common compile time errors:

„Test::x" is not accessible
„Test::y" is not accessible

Protected access is the mechanism same as private. It is frequently used in inheritance. Private members are not inherited at any case whereas protected members are inherited. The keyword **private** followed by colon (:) is used to make data member and member function visible with in the class and its child classes.

What is a member function? How can you declare member functions?

Member functions are defined in two places. They are;

1. Inside the class
2. Outside the class

Inside the class

Member functions can be defined immediately after the declaration inside the class. These functions by default act as **inline functions**.

Example:

```
class student
{
      private:
            int rno;
           char *sname;
      public:
```

```
        void print()
        {
                cout<<"rno="<<rno;
                cout<<"name="<<sname;
        }
        void read(int a, char *s)
        {
                rno = a;
                strcpy(snm, s);
        }
};
void main()
{
        student s;
        s.read(10, "Rama");
        s.print();
}
```

Output:
rno= 10
name = Rama

Outside the class

- ✓ Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions.
- ✓ The main difference is the member function incorporates a membership identity label in the header. This label tells the compiler which class the function belongs to.

General form

```
return-type classname :: function-name (list of arguments)
{
        // function body
}
```

Example:

```
class student
{
        private:
                int rno;
                char *sname;
        public:
                void read(int , char*);
                 void print();
};
```

```
void student :: read (int a, char *s)
{
      no = a;
      strcpy(sname, s);
}
void student :: print()
{
      cout<<"rno="<<rno;
      cout<<"name="<<sname;
}
void main()
{
      student s;
      s.read(10, "Rama");
      s.print();
}
```

```
Output:
rno= 10
name = Rama
```

What are the characteristics of member functions?

- ✓ The member functions are accessed only by using the object of the same class.
- ✓ The same function can be used in any number of classes.
- ✓ The private data or private functions can be accessed only by public member functions.

How can you declare outside member function as inline? Explain with an example. (OR) Explain about inline keyword.

- ✓ Inline function mechanism is useful for small functions.
- ✓ The inline functions are similar to macros. By default, all the member functions defined inside the class are inline functions.
- ✓ The member functions defined outside the class can be made inline by prefixing "inline" to the function declaration.

General form:

```
inline Return-type classname :: function-name (list of
arguments)
{
      //function body
}
```

Example:

```
class student
{
      private:
            int rno;
            char *sname;
```

```
public:
void print();
void read(int    a, char *s)
        {
                    no = a;
                    strcpy(snm, s);
              }
};
inline void student :: print()
{
        cout<<"no="<<no;
        cout<<"name="<<snm;
}
void main()
{
        student s;
         s.read(10, "Rama");
        s.print();
}
```

Output:
rno= 10
name = Rama

What is meant by data hiding? (OR) How data hiding is accomplished in C++? (OR) Explain about classes. (OR) Explain about data encapsulation.

Data hiding is also called as data encapsulation. An encapsulated object is called as abstract data type. Data hiding is useful for protecting data. It is achieved by making the data variables of class as private. Private variables cannot directly accessible to the outside of the class. The keywords private and protected are used to protect the data.

Access Specifier	**Members of the Class**	**Class Objects**
Public	Yes	Yes
Private	Yes	No
Protected	Yes	No

Example:

```
class ex
{
        private:
                int p;
        protected:
                int q;
        public:
               int r;
        void getp()
```

```
    {
        p=10;
        cout<<"private="<<p;
    }
    void getq()
    {
        q=10;
        cout<<"protected="<<q;
    }
    void getr()
    {
        r=10;
        cout<<"public ="<<r;
    }
};
void main()
{
    ex e;
    e.getp();
    e.getq();
    e.getr();
    e.r=100;
    e.getr();
}
```

Output:
private =10
protected=10
public =10
public =100

How can you overload a member function? Explain with an example.

Member functions can be overloaded like any other normal functions. Overloading means one function is defined with multiple definitions with same functions name in the same scope.

The following program explains the overloaded member functions

```
class absv
{
    public:
        int num(int i);
          double num( double d)
};
```

```
int absv:: num(int i)
{
      return (abs(i));
}
double absv:: num( double d)
{
      return (fabs( d));
}
void main()
{
      absv a;
      cout<<"\nThe absolute value of -10 is "<<n. num(-10);
      cout<<"\nThe absolute value of -12.35 is "<<n.num(-12.35);
}
```

Output:
The absolute value of -10 is 10
The absolute value of -12.35 is 12.35

What is a nested class? Explain with an example.
When a class defined in another class, it is known as nesting of classes. In nested classes, the scope of inner class is restricted by outer class

The following program illustrates the nested classes:

```
class A
{
public :
      class B
      {
            void show()
            {
                  cout<<"\nC++ is wonderful language";
            }
       };
};

int main(void)
{
      A::B x;
      x.show();
}
```

Output:
C++ is wonderful language

What is a constructor? What is a destructor? What are their properties?

A constructor is a special member function used for automatic initialization of an object. Whenever an object is created, the constructor is called automatically. Constructors can be overloaded. Destructor destroys the object. Constructors and destructors having the same name as class, but destructor is preceded by a tilde (~) operator. The destructor is automatically executed whenever the object goes out of scope.

Characteristics of C*onstructors*

- ✓ Constructors have the same name as that of the class they belongs to.
- ✓ Constructors must be declared in public section.
- ✓ They automatically execute whenever an object is created.
- ✓ Constructors will not have any return type even void.
- ✓ Constructors will not return any values.
- ✓ The main function of constructor is to initialize objects and allocation of memory to the objects.
- ✓ Constructors can be called explicitly.
- ✓ Constructors can be overloaded.
- ✓ A constructor without any arguments is called as default constructor.

What are the applications of constructors? (Default constructor)

Constructors are used to initialize member variables of a class. Constructors allocate required memory to the objects. Constructors are called automatically whenever an object is created. A constructor which is not having any arguments are said to be default constructor.

Example:

```
class num
{
      int a, b;
    public:
      num()
      {
            a =5; b=2;
      }
      void show()
      {
            cout<<a<<b;
      }
};
```

```
Void main()
{
      num n;
      n.show();
}
```

Output:
5 2

Explain about parameterized constructors with an example.

- ✓ It may be necessary to initialize the various data elements of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created.
- ✓ The constructors that can take arguments are called parameterized constructors.
- ✓ They can be called explicitly or implicitly.

Example:

```
class num
{
      int a, b ,c;
    public:
      num(int x, int y)
      {
            a=x; b=y;
      }
      void show()
      {
            cout<<a<<b;
      }
};
void main()
{
      num n(10,20);
      //implicit call
      n.show();
      num x= num(1,2);
      //explicit call x.show();
}
```

Output:
10 20
1 2

What is meant by constructor overloading? Explain with an example. (Multiple Constructors)

- ✓ Similar to normal functions, constructors also overloaded. C++ permits to use more than one constructors in a single class.
- ✓ If a class contains more than one constructor. This is known as

constructor overloading.

Add( ) ; // No arguments
Add (int, int) ; // Two arguments

Example:

```
class num
{
      int a, b;
    public:
      num()       // Default constructor
      {
            a =10; b=20;
      }
      num(int x, int y)// Parameterized constructor

      {
            a=x;
            b=y;
      }
      void add()
      {
            cout<<a+b;
      }
};
void main()
{
      num n;
      n.add();
      num x(1,2);
      x.add();
}
```

Output:
30
3

The num class has two constructors. They are;

- ✓ Default constructor
- ✓ Parameterized constructor

Whenever the object "n" is created the default constructor is executed automatically and a,
and b values are initialized to 10, 20 respectively.

Whenever the object "x" is created the parameterized constructor is executed automatically and a, and b values are assigned with 1 and 2 respectively.

Explain about constructors with default argument with an example.

- ✓ Similar to functions, It is possible to define constructors with default arguments.
- ✓ Consider power(int n, int p= 2);
  - The default value of the argument p is two.
  - power p1 (5) assigns the value 5 to *n* and 2 to *p*.
  - power p2(2,3) assigns the value *2* to n and 3 to *p*.

Example:

```
class power
{
      int b,p;
    public:
      power(int n=2, int m=3)
      {
            b=n;
            p=m;
            cout<<pow(n,m);
      }
};
void main()
{
      power x;
      Power y(5);
      power z(3,
      4);

}
```

Output:
8 125 81

Explain about copy constructor with an example.

- ✓ Copy constructor is used to declare and initialize an object from another object.
- ✓ A copy constructor takes a reference to an object of the same class as itself as an argument.

Ex:

```
class Test
{
      int i;
    public:
```

```
        Test()              // Default constructor
        {
              i=0;
        }
        Test (int a)        // Parameterized constructor
        {
              i = a;
        }
        Test (code &x)    //Copy Constructor
        {
              i = x.i;
        }
        void show()
        {
        cout<<i<<endl;
        }
};
void main()
{
        Test a(100);
        a.show();
        Test b(a);          //Copy Constructor
        invoked b.show();
}
```

Output:
100 100

Explain about Destructors with an example.

- ✓ Destructor is a special member function like a constructor. Destructors destroy the class objects that are created by constructors.
- ✓ The destructor have the same name as their class, preceded by a ~.
- ✓ The destructor neither requires any arguments nor returns any values.
- ✓ It is automatic ally executed when the object goes out of score.
- ✓ Destructor releases memory space occupied by the objects.

Characteristics of Destructors

- ✓ Their name is the same as the class name but is preceded by a tilde(~).
- ✓ They do not have return types, not even void and they cannot return values.
- ✓ Only one destructor can be defined in the class.
- ✓ Destructor neither has default values nor can be overloaded.
- ✓ We cannot refer to their addresses.
- ✓ An object with a constructor or destructor cannot be used as a member of a union.

- ✓ They make "implicit calls" to the operators ***new*** and ***delete*** when memory allocation/ memory de-allocation is required.
- ✓

Example:

```
class Test
{
   public:
      Test()
      {     cout<<"\n Constructor Called";

      }
      ~Test()
      {
            cout<<"\n Destructor Called";
      }
};
void main()
{
      Test t;
}
```

Output:
Constructor Called
Destructor Called

What is meant by anonymous objects? Explain.

It is possible to declare objects without any name. These objects are said to be anonymous objects. Constructors and destructors are called automatically whenever an object is created and destroyed respectively. The anonymous objects are used to carry out these operations without object.

Ex:

```
class noname
{
      int x;
    public:
      noname()
      {
            cout<<"\n In Default Constructor";
            x=10;
            cout<<x;
      }
      noname(int i)
      {
            cout<<"\n In Parameterized Constructor";
            x=i;
            cout<<x;
      }
      ~noname()
      {
            cout <<"\n In Destructor";
      }
};
void main()
{
      noname();
      noname(12);
}
```

Output:
In Default Constructor 10
In Parameterized
Constructor 12 In
Destructor
In Destructor

UNIT III – Syllabus

Operator Overloading and Type Conversion & Inheritance: The Keyword Operator, Overloading Unary Operator, Operator Return Type, Overloading Assignment Operator (=), Rules for Overloading Operators, Inheritance, Reusability, Types of Inheritance, Virtual Base Classes- Object as a Class Member, Abstract Classes, Advantages of Inheritance, Disadvantages of Inheritance.

What is meant by operator overloading? Explain with an example.

A symbol that is used to perform an operation is called an operator. It is used to perform operations on constants and variables. By using operator overloading, these operations are performed on objects. It is a type of polymorphism. The keyword **operator** is used to define a new operation for an operator.

Syntax:

```
return-type operator operator-symbol (list of arguments)
{
        // Set of statments
}
```

Steps:

1. Define a class which is to be used with overloading operators.
2. Declare the operator prototype function is in public section.
3. Define the definition of the operator.

Example

```
# include <iostream.h>
# include <conio.h>
class number
{
   public:
        int x,y;
        number()
        {
                x=0;
                y=0;
        }
        number(int a, int b)
        {
                x=a;
                y=b;
        }
        number operator + (number d)
        {
                number t;
                t.x=x+d.x;
                t.y=y+d.y;
                return t;
        }
        void show()
        {
                cout<<"\nX="<<x<<"\t Y="<<y;
        }
};

int main()
{
        number n1(10,20), n2(20,30), n3;
        n3=n1+n2;
        n3.show();
        return 0;
}
```

Output X=10 Y=20 X=20 Y=30 X=30 Y=40

Explain about overloading unary operators with an example.

- An operator whichtakes only one argument is called as unary operator.
  Ex: ++, --, -, +, !, ~, etc.
- A class member function will take zero arguments for overloading unary operator.
- A friend member function will take only one argument for overloading unary operator.

Constraints on increment /decrement operators

When ++ and − − operators are overloaded, there exists no difference between the postfix and prefix overloaded operator functions. To make the distinction between prefix and postfix

notations of operator, a new syntax is used to indicate postfix operator overloading funct ion. The syntaxes are as follows:

Operator ++(int); //postfix notation
Operator ++( ); //prefix notation

Ex: Program for overloading unary operator using normal member function.

```
# include <iostream.h>
# include <conio.h>
class num
{
    private:
        int a, b;
    public:
        num(int x, int y)
        {
                a=x;
                b=y;
        }
        void show()
        {
                cout<<"\nA="<<a<<"\tB="<<b;
        }
        void operator ++( )             //prefix notation
        {
                ++a;
                ++b;
        }
        void operator --(int)           //postfix notation
        {
                a--;
                b--;
        }
};

int main()
{
        num x(4,10);
        x.show();
        ++x;
        cout<<"\n After increment";
        x.show();
        x--;
        cout<<"\n After decrement";
        x.show();

}
```

Output

After increment
A = 5 B = 11
After decrement
A = 4 B = 10

Ex: Program for overloading unary operator using friend function.

```
# include <iostream.h>
# include <conio.h>
class complex
{
    private:
        int real, imag;
    public:
        complex()
        {
                real=imag=0;
        }
        complex(int r, int i)
        {
                real=r;
                imag=i;
        }
        void show()
        {
                cout<<"\n real="<<real<<"\timaginary="<<imag;
        }
        friend complex operator -(complex &c)
        {
        =-c.r;
                c.i=-c.i;
                return c;
        }
};
void main()
{
        complex c(1,-2);
           how();
        cout<<"\nAfter sign change";
        -c;
        c.show();
}
```

Output

```
real= 1         imaginary=-2
After sign change
real= -1        imaginary=2
```

Explain about overloading binary operators with an example.

- An operator which takes two arguments is called as binary operator.
  Ex: +, *, /, etc.
- A class member function will take one argument for overloading binary operator.
- A friend function will takes two arguments for overloading binary operator.

Example: Program for overloading binary operator using class member function.

```
# include <conio.h>
# include <iostream.h>
class num
{
    private:
```

```
        int a , b;
    public:
        void input()
        {
                cout<<"\nEnter two numbers:";
                cin>>a>>b;
        }
        void show()
        {
                cout<<"\nA= "<<a<<"\tB=  "<<b;
        }
        num operator +(num n)
        {
                num t;
                t.a=a+n.a;
                t.b=b+n.b;
                return t;
        }
        num operator -(num n)
        {
                num t;
                t.a=a-n.a;
                t.b=b-n.b;
                return t;
        }
};
int main()
{
        num x,y,z;
        x.read();
        y.read();
        z=x+y;
        r.show();
        return 0;
}
```

Output

```
Enter two numbers: 1 2
Enter two numbers: 3 4
A=4     B=6
```

Ex: Program for overloading binary operator using friend function.

```
# include <conio.h>
# include <iostream.h>
class num
{
    private:
        int a, b;
    public:
        void input()
        {
                cout<<"\nEnter two numbers:";
                cin>>a>>b;
        }
        void show()
        {
                cout<<"\nA="<<a<<"\tB="<<b;
        }
        friend num operator + (num o1, num o2)
        {
                num t;
                t.a=o1.a+o2.a;
                t.b=o1.b+o2.b;
                return t;
        }
};
int main()
{
        num x,y,z;
        x.input();
        y.input();
        z=x+y;
        z.show();
        return 0;
}
```

Output

```
Enter two numbers: 5 8
Enter two numbers: 1 4
A=6     B=12
```

Explain about overloading assignment operator with an example.

Data members of one object are initialized with some values, and same values are assigned to another object with assignment operator. Assignment operators can be overloaded in two ways. They are:

1. Implicit overloading
2. Explicit overloading

Implicit overloading:

```
# include <iostream.h>
class num
{
    private:
        int x;
    public:
```

```
        num(int a)
        {
                x=a;
        }
        void show()
        {
                cout<<x<<"  ";
        }
};
int main()
{
        num n1(2), n2(3);
        cout<<"\nBefore assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        a2=a1;          //Implicit assigment
        cout<<"\nAfter assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
}
```

Output

Before assignment:
A = 2 B=3
After assignment:
A = 2 B=2

Explicit overloading:

```
# include <iostream.h>
class num
{
        private:
           int x;
        public:
        num(int a)
        {
                x=a;
        }
        void show()
        {
                cout<<X<<"  ";
        }
        void operator =(num b)
        {
                x=b.x;
        }
};
int main()
{
        num a1(2), a2(3);
```

```
        num n1(2), n2(3);
        cout<<"\nBefore assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        a1.operator=(a2);                    //Explicit assigment
        cout<<"\nAfter assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        return 0;
}
```

Output

Before assignment:
A = 2 B=3
After assignment:
A = 3 B=3

Explain about rules for overloading operators.

1. Operator overloading can't change the basic idea.
2. Operator overloading never changes its natural meaning. An overloaded operator "+" can be used for subtraction of two objects, but this type of code decreased the utility of the program.
3. Only existing operators can be overloaded.
4. The following operators can't be overloaded with class member functions

Operator	Description
.	Member operator
.*	Pointer to member operator
::	Scope resolution operator
sizeof()	Size of operator
# and ##	Preprocessor symbols

5. The following operators can't be overloaded using friend functions.

Operator	Description
( )	Function call operator
=	Assignment operator
[ ]	Subscripting operator
–>	Class member access operator

6. In case of unary operators normal member function requires no parameters and friend function requires one argument.
7. In case of binary operators normal member function requires one argument and friend function requires two arguments.
8. Operator overloading is applicable only within in the scope.
9. There is no limit for the number of overloading for any operation.
10. Overloaded operators have the same syntax as the original operator.

What is inheritance? What are the advantages and disadvantages of inheritance?

(OR)

Explain about reusability.

Inheritance is the most important and useful feature of OOP. Reusability can achieved with the help of inheritance. The mechanism of deriving new class from an old class is called as inheritance. The old class is known as *parent class* or *base class*. The new one is called as *child class* or *derived class*. In addition to that properties new features can also be added.

Advantages:

- Code can be reused.
- The derived class can also extend the properties of base class to generate more dominant objects.
- The same base class is used for more derived classes.
- When a class is derived from more than one class, the derived classes have similar properties to those of base classes.

Disadvantages:

- Complicated.
- Invoking member functions creates overhead to the compiler.
- In class hierarchy, various data elements remains unused, and the memory allocated to them is not utilized.

What are access specifiers? How class are inherited?

C++ provides three different access specifiers. They are:

1. Public
2. private and
3. protected.

- The public data members can be accessed directly outside ofthe class with the object.
- The private members are accessed by the public member functions of the class.
- The protected members are same as private but only the difference is protected members are inherited while private members are not inherited.

A new class can be derived from the old one is as follows:

```
class name_of_the_derived_class : access_specifier name_of_the_base_class
{
        ……
        Members of the derived class
        ......
};
```

Example:

```
class A : public B
{
        ..............
        ..............
};

class A : private B
{
..............
..............
```

};
class A : protected B
{
.....
.....
};

Note: If no access specifier is specified then by default it will takes as private.

Type of inheritance	Base class member	Derived class member
Private	Private	Not accessible
	Public	Private
	Protected	Private
Public	Private	Not accessible
	Public	public
	Protected	Protected
Protected	Private	Not accessible
	Public	Protected
	Protected	Protected

Public derivation:
In public derivation, all the public members of base class become public members of the derived class and protected members of the base class becomes protected members to the derived class. Private members of the base class will not be inherited.

Example:

```
# include <iostream.h>
class Base
{
        public: int x;
};
class Derived : public Base
{
        public: int y;
};
int main()
{
        Derived d;
        d.x=10;
        d.y=20;
        cout<<"\n Member x="<<b.x;
        cout<<"\n Member y="<<b.y;
        return 0;

}
```

Output:
Member x=10
Member y=20

Private derivation

In private derivation, all the public and protected members of the base class become private members of the derived class and private members of the base class will not be inherited.

Example:

```
# include <iostream.h>
class Base
{
        public: int x;
};

class Derived : private Base
{
        int y;
    public:
        Derived( )
        {
                x=10;
                y=20;
        }
        void show()
        {
                cout<<"\n Member x="<<x;
                cout<<"\n member y="<<y;
        }
};
int main()
{
        Derived d;
        d.show():
        return 0;
}
```

Output:

```
Member x=10
Member y=20
```

Protected derivation

In protected derivation, all the public and protected members of the base class become protected members of the derived class and private members of the base class will not be inherited.

Example:

```
# include <iostream.h>
class Base
{
        public: int x;
};

class Derived : protected Base
{
        int y;
    public:
```

```
        Derived( )
        {
                x=10;
                y=20;
        }
        void show( )
        {
                cout<<"\n Member x="<<x;
                cout<<"\n member y="<<y;
        }
};

int main()
{
        Derived d;
        d.show();
}
```

Output:
Member x=10
Member y=20

Explain about protected data with private members with an example.
The protected type is similar to private, but it allows the derived class to access the members of the protected.

Example

```
# include <iostream.h>
class Base
{
        protected: int x;
};

class Derived : private Base
{
    int y;
        public:
          Derived( )
          {
                x=10;
                y=20;
          }
          void show()
          {
                cout<<"\n Member x="<<x;
                cout<<"\n Member y="<<y;
          }
};
int main()
{
        Derived d;
        d.show();
}
```

Output:
Member x=10
Member y=20

Define inheritance. Explain about various types of inheritance with examples.
Deriving a new class from an existing one is called as inheritance.
Types:

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance
6. Multipath Inheritance

Single Inheritance:
In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance..

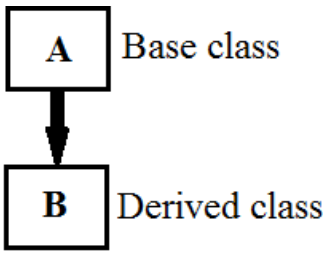


Fig. Single Inheritance

Here A is the base class and B is the derived class.

Example:

```
#include<iostream.h>
#include<conio.h>
class Emp
{
  public:
    int eno;
    char ename[20],desig[20];
    void input()
    {
          cout<<"Enter employee no:";
          cin>>eno;
          cout<<"Enter employee name:";
          cin>>ename;
          cout<<"Enter designation:";
          cin>>desig;
    }
};

class Salary: public Emp
{
    float bp, hra, da, pf, np;
  public:
    void input1()
    {
          cout<<"Enter Basic pay:";
          cin>>bp;
          cout<<"Enter House Rent Allowance: ";
          cin>>hra;
          cout<<"Enter Dearness Allowance :";
          cin>>da;
          cout<<"Enter Provident Fund:";
          cin>>pf;
    }
    void calculate()
    {
          np=bp+hra+da-pf;
    }
    void display()
    {
          cout<<"\nEmp no: "<<eno
          cout<<"\nEmp name: "<<ename;
          cout<<"\nDesignation: "<<design;
          cout<<"\nBasic pay:"<<bp;
          cout<<"\nHRA:"<<hra;
          cout<<"\nDA:"<<da;
          cout<<"\nPF:"<<pf;
          cout<<"\nNet pay:"<<np;
    }
```

```
};

int main()
{
   clrscr();
   Salary s;
   s.input();
   s.input1();
   s.calculate();
   s.show();
   getch();
   return 0;
}
```

Output:

```
Enter employee number: 1001
Enter employee name: Vijayanand
Enter designation: Manager
Enter basic pay:25000
Enter House Rent Allowance:2500
Enter Dearness Allowance :5000
Enter Provident Fund:1200

Emp no: 1001
Emp name: Vijayanand
Designation: Manager
Basic pay:25000
HRA:2500
DA:5000
PF:1200
Net pay: 31300
```

Multiple Inheritance:

In this type of inheritance a class may derive from two or more base classes.

(or)

When a class is derived from more than one base class, is called as multiple inheritance.

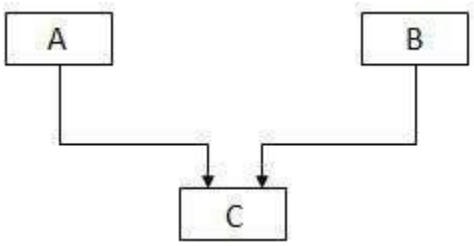


Fig. Multiple Inheritance

Where class *A* and *B* are Base classes and *C* is derived class.

Example:

```
#include<iostream.h>
#include<conio.h>
class Student
{
      protected:
        int rno,m1,m2;
      public:
        void input()
        {
                cout<<"Enter the Roll no :";
                cin>>rno;
                cout<<"Enter the two subject marks :";
                cin>>m1>>m2;
        }
};

class Sports
{
     protected:
        int sm;                    // sm = Sports mark
     public:
```

```
        void getsm()
        {
            cout<<"\nEnter the sports mark :";
            cin>>sm;
        }
};
class Report : public student, public sports
{
        int tot, avg;
    public:
        void show()
        {
            tot=(m1+m2+sm);
            avg=tot/3;
            cout<<"\nRoll No : "<<rno<<"\nTotal : "<<tot;
            cout<<"\n\tAverage : "<<avg;
        }
};

int main()
{
        clrscr();
        Report r;
        r.input();
        r.getsm();
        r.show();
        return 0;
}
```

Output:
Enter the roll no: 10
Enter the two marks : 70 90
Enter the sports mark: 60
Roll no : 10
Total : 110
Average: 73

Hierarchical Inheritance
In this type of inheritance, multiple classes are derived from a single base class.

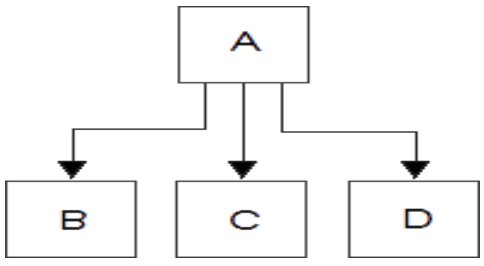


Fig. Hierachical Inheritance

Where class *A* is the base class and *B, C* and *D* are derived classes.
Example:

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

```
class Vehicle
{
    public:
        Vehicle()
        {
            cout<<"\nIt is motor vehicle";
        }
};
class TwoWheelers : public Vehicle
{
    public:
        TwoWheelers()
        {
            cout<<"\nIt has two wheels";
        }
        void speed()
        {
                cout<<"\nSpeed: 80 kmph";
        }
};
class ThreeWheelers : public Vehicle
{
    public:
        ThreeWheelers()
        {
            cout<<"\nIt has three wheels";
        }
        void speed()
        {
                cout<<"\nSpeed: 60 kmph";
        }
};
class FourWheelers : public Vehicle
{
    public:
        FourWheelers()
        {
            cout<<"\nIt has four wheels";
        }
        void speed()
        {
                cout<<"\nSpeed: 120 kmph";
        }
};

int main( )
{
        clrscr();
        TwoWheelers two;
        two.speed();
        cout<<"\n--------------------------";
        ThreeWheelers three;
        three.speed();
```

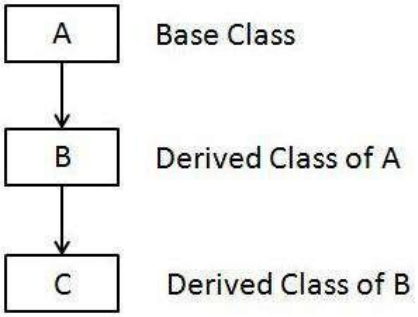
```
        cout<<"\n--------------------------";
        FourWheelers four;
        four.speed();
        getch();
        return 0;
}
```

Output

```
It is motor vehicle
It has two wheels
Speed: 80 kmph";
-----------------------------
It is motor vehicle
It has three wheels
Speed: 60 kmph";
-----------------------------
It is motor vehicle
It has four wheels
Speed: 120 kmph";
```

Multilevel Inheritance

In this type of inheritance, the derived class inherits from a class, which in turn inherits from some other class.



Where *class A* is the base class, *C* is derived class and *B* acted as base class as well as derived class.

Example:

```
#include<iostream.h>
#include<conio.h>
class Car
{
    public:
        Car()
        {
            cout<<"\nVehicle type: Car";
        }
};
class Maruti : public Car
{
    public:
        Maruti()
        {
           cout<<"\nComnay: Maruti";          }
        }
        void speed()
        {
```

```
                cout<<"\nSpeed: 90 kmph";
        }
};
class Maruti800 : public Maruti
{
    public Maruti800()
   {
       cout<<"\nModel: Maruti 800");
   }
   void speed()
   {
        cout<<"\nSpeed: 120 kmph";
   }
};
int main( )
{
        clrscr();
        Maruti800 m;
        m.speed();
        getch();
}
```

Output:
Vehicle type: Car
Company: Maruti
Model: Maruti 800
Speed: 120K mph

Hybrid (Virtual) Inheritance
Hybrid Inheritance is combination of one or more types of inheritance.

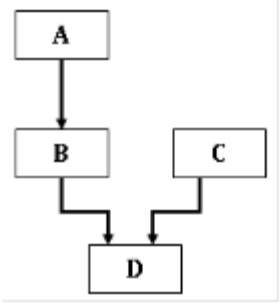


Fig. Hybrid Inheritance

Where class A to class B forms single inheritance, and class B,C to class D form Multiple inheritance.

Example:

```
#include<iostream.h>
#include<conio.h>
class Player
{
    protected:
        char name[20];
        char gender;
        int age;
};

class Physique : public Player
{
    protedted:
        float height,weight;
};
class Location
{
     protected:
        char city[15];
        long int pin;
};
class   Game    : public Physique, public
Location
```

```
{
        char game[15];
    public:
        void input()
         {
            cout<<"\nEnter Player Information";
            cout<<"Name: ";
            cin>>name;
            cout<<"Genger: ";
            cin>>gender;
            cout<<"Age: ";
            cin>>age;
            cout<<"Height and Weight: ";
            cin>>heigh>>weight;
            cout<<"City: ";
            cin>>city;
            cout<<"Pincode: ";
            cin>>pin;
            cout<<"Game played: ";
            cin>>game;

     }
     void input()
     {
            cout<<"\nPlayer Information";
            cout<<"\nName: "<<name;
            cout<<"\nGenger: "<<gender;
            cout<<"\nAge: "<<age;
            cout<<"\nHeight<<height;
            cout<<"\nWeight: "<<weight;
            cout<<"\nCity: "<<city;
            cout<<"\nPincode: "<<pin;
            cout<<"\nGame: "<<game;
      }
};
int main( )
{
        clrscr();
        Game  g;
        g.input();
        g.show();
        return 0;
}
```

Output

```
Enter Player Information
Name: Azar
Genger: M
Age: 38
Height and Weight: 5.8 70
City: Hyderabad
Pincode: 522183
Game played: Cricket

Player Information
Name: Azar
Genger: M
Age: 38
Height and Weight: 5.8 70
City: Hyderabad
Pincode: 522183
Game played: Cricket
```

Multi-path Inheritance

In this, one class is derived from two base classes and in turn these two classes are derived from a single base class in known as Multi-path Inheritance**.**

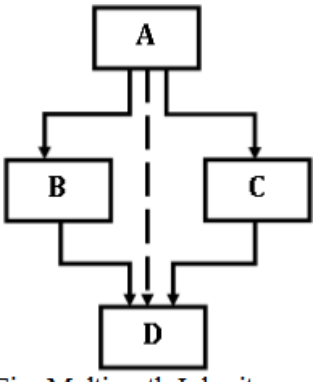


Fig. Multi-path Inheritance

Example

```
class A
{
        //class A definition
};
class B: public A
{
        //class B definition
};
class C: public A
{
        //class C definition
};
class D :public B, public  C
{
        //class D definition
```

```
};
```

Explain about virtual base classes with an example.

(OR)

How can you overcome the ambiguity occurring due to multipath inheritance? Explain with an example.

To overcome the ambiguity due to multipath inheritance the keyword virtual is used. When classes
are derived as virtual, the compiler takes essential caution to avoid the duplication of members.

Uses:

When two or more classes are derived from a common base class, we can prevent multiple copies of
the base class in the derived classes are done by using **virtual** keyword. This can be achieved by
preceding the keyword "**virtual"** to the base class.

Example

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

class A
{
        protected:
                int a1;
};
class B: public virtual A
{
        protected:
                int a2;

};
class C: public virtual A
{
        protected:
                int a3;

};
class D :public B, public C
{
        int a4;
    public:
        void input()
        {
            cout<<"Enter a1,a2,a3 and a4 values:";
            cin>> a1>>a2>>a3>>a4;
        }
        void show()
        {
            cout<<"a1="<<a1<<"\na2="<<a2;
            cout<<"\na3="<<a3<<"\na4="<<a4;
        }
```

```
};
int main()
{
    D d;
    d.input();
    d.show();
    return 0;
}
```

Output

```
Enter a1, a2, a3 and a4 values: 10 20 30 40
a1=10
a2=20
a3=30
a4=40
```

How constructors and destructors are executed in inherited class? Explain with an example.

The constructors are used to initialize the member variables and the destructors are used to destroy the object. The compiler automatically invokes the constructor and destructors.

Rules:

- The derived class does not require a constructor if the base class contains default constructor.
- If the base class is having a parameterized constructor, then it is necessary to declare a constructor in derived class also. The derived class constructor passes arguments to the base class constructor.

Example:

```
#include<iostream.h>
class A
{
    public:
        A()
        {
                cout<<"\n Class A constructor called";
        }
        ~A()
        {
                cout<<"\nClass A destructor called";
        }
};

class B : public A
{
    public:
        B()
        {
                cout<<"\n Class B constructor called";
```

```
        }
        ~B()
        {
                cout<<"\nClass B destructor called";
        }
};

class C : public B
{
    public:
        C()
        {
                cout<<"\n Class C constructor called";
        }
        ~C()
        {
                cout<<"\nClass C destructor called";
        }
};

int main()
{
        C c;
        return 0;
}
```

Output

```
Class A constructor called
Class B constructor called
Class C constructor called
Class C destructor called
Class B destructor called
Class A destructor called
```

How can you pass an object as a class member? Explain.

(OR)

Explain about delegation with an example.

(OR)

Explain about container classes with an example.

Declaring the object as a class data member in another class is known as delegat ion. When a class has an object of another class as its member, such a class is known as a container class. This kind of relationship is known as has-a-relationship or containership.

Example

```
# include <iostream.h>
class A
{
     public:
        int x;
        A()
        {
                x=20;
                cout<<"\n In A constructor";
```

```
        }
};

class B
{
    public:
        int y;
        A a;
        B()
        {
                y=30;
                cout<<"\n In B constructor";
        }
        void show()
        {
                cout<<"\n X="<<a.x<<"\t Y="<<y;
        }
};
int main()
{
        B b;
        b.show();
        return 0;
}
```

Output

```
In A constructor
In B constructor
X=20 Y=30
```

Define abstract class. What is the use of abstract classes? Explain.

An abstract class is a class not used for creating objects. It is designed only to act as a base class. These classes are similar to a skeleton on which new classes are designed. These classes contain pure virtual functions.

Example

```
#include <iostream.h>
class Shape
{
   protected:
        int width;
        int height;
  public:
        virtual int getArea() = 0;
        void setWidth(int w)
        {
                width = w;
        }
        void setHeight(int h)
        {
                height = h;
        }
 };
```

```
class Rectangle: public Shape
{
  public:
        int getArea()
        {
                return (width * height);
        }
};
class Triangle: public Shape
{
    public:
        int getArea()
        {
                return (width * height)/2;
        }
};

void main()
{
        Rectangle r;
        Triangle t;
        r.setWidth(5);
        r.setHeight(7);
        cout << "\nRectangle area: " << r.getArea() << endl;
        t.setWidth(5);
        t.setHeight(7);
        cout << "\nTriangle area: " << t.getArea() << endl;
      return 0;
}
```

Output:
Rectangle area : 35
Triangle area : 17

UNIT-IV

Pointers

A pointer is a variable which is used to store address of another variable.i.e pointer value is an address.
Syntax is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable.
Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

```
int   *ip;   // pointer to an integer
double *dp;   // pointer to a double
float  *fp;   // pointer to a float
char   *ch    // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Example-1:

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int *  p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

Example-2 : Swap two numbers using pointers

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
  return 0;
}
```

Features of Pointers

- Pointers save memory space.
- Execution time with pointers is faster because data are manipulated with the address, that is, direct access tomemory location.
- Memory is accessed efficiently with the pointers. The pointer assigns and releases the memory as well. Hence it can be said the Memory of pointers is dynamically allocated.
- Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensionalarrays.
- An array, of any type can be accessed with the help of pointers, without considering its subscript range.
- Pointers are used for file handling.
- Pointers are used to allocate memory dynamically.
- In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.

Pointer to class

To access members of a class using pointer we have to use the member access operator -> operator.

We can **define pointer of class** type, which can be used to point to class objects.

Example-1:

```
class Simple
{
   public:
   int a=5;
};
int main()
{
   Simple obj;
   Simple* ptr;   // Pointer of class type
ptr = &obj;

cout<<obj.a;
cout<<ptr->a;  // Accessing member with pointer
}
```

Example-2

```
#include <iostream>
using namespace std;
class Box {
private:
    double length;
    double breadth;
    double height;
```

```
  public:
    // Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0) {
cout<<"Constructor called." <<endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }

};

int main() {
  Box Box1(3.3, 1.2, 1.5);   // Declare box1
  Box Box2(8.5, 6.0, 2.0);   // Declare box2
  Box *ptrBox;               // Declare pointer to a class.

  // Save the address of first object
ptrBox = &Box1;

  // Now try to access a member using member access operator
cout<< "Volume of Box1: " <<ptrBox->Volume() <<endl;

  // Save the address of second object
ptrBox = &Box2;

  // Now try to access a member using member access operator
cout<< "Volume of Box2: " <<ptrBox->Volume() <<endl;

  return 0;
}
```

Pointer Objects

When an object is a pointer to the class, the member functions of that class is accessed by using an object of that class with an arrow (->) operator.

Let us consider an example:

```
#include <iostream>
using namespace std;
class Sample { // This a class Sampleprivate:
            int value1, value2;
      public:
            void setValues(int num1, int num2) {
                  value1 = num1;
                  value2 = num2;
            }
            void display() {
                  cout<< "The given values are : " << value1 << " " << value2 <<endl;
            }};
```

```
int main() {
        Sample *p = new Sample; // A pointer object  p is created and stored the address of the
//dynamically allocated memory of a class Sample
        p->setValues(88, 99); // Pointer object p can access the public member functions using ->//operator
        p->display();
}
```

In the above example, a pointer object is referenced to the class Sample and the pointer object can access its public members by using arrow (->) operator.

Pointers to base class

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.

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

class Rectangle: public Polygon {
 public:
  int area()
 { return width*height; }
};

class Triangle: public Polygon {
 public:
  int area()
 { return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * p1 = &rect;
  Polygon * p2 = &trgl;
  p1->set_values (4,5);
  p2->set_values (4,5);
cout<<rect.area() << '\n';
cout<<trgl.area() << '\n';
  return 0;
}
Output:
20
10
```

Function main declares two pointers to Polygon (named p1 and p2). These are assigned the addresses of rect and trgl, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.

Dereferencing p1 and p2 (with p1-> and p2->) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:

```
p1->set_values (4,5);
rect.set_values (4,5);
```

But because the type of both p1 and p2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

Pointer to Derived Class

In C++, we can declare a pointer points to the base class as well as derive class.
Consider below example to understand pointer to derived class.

```
#include<iostream.h>
class base
{
   public:
   int n1;
   void show()
   {
cout<<"\nn1 = "<<n1;
   }
};
class derive : public base
{
   public:
   int n2;
   void show()
   {
cout<<"\nn1 = "<<n1;
cout<<"\nn2 = "<<n2;
   }
};


int main()
{
  base b;
  base *bptr;
bptr=&b;        //address of base class
cout<<"Pointer of base class points to it";
bptr->n1=44;     //access base class via base pointer
bptr->show();
  derive d;
cout<<"\n";
bptr=&d;        //address of derive class
```

```
    bptr->n1=66;          //access derive class via base pointer
    bptr->show();  //executes base show only
      derive *d1;//derived pointer
      d1=&d;
    // d1=&b;// error: invalid conversion from 'base*' to 'derive*'
    cout<<"\npointer to derived class";
      d1->n1=100;
      d1->n2=200;
      d1->show();
      return 0;
    }
```

Output

```
Pointer of base class points to it
n1 = 44
n1 = 66
pointer to derived class
n1 =100
n2 =200
```

Here the show() method is the overridden method, bptr execute show() method of 'base' class twice and display its content. Even though bptr first points to 'base' and second time points to 'derive' ,both the time bptr->show() executes the 'base' method show().
If we create pointer for derived class and assign derived class object then we access derived class members.

this pointer

First we have to know how objects look at functions and data members of a class is each object gets its own copy of data members and all objects share a single copy of member functions.

Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as 'this'.
The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

Every object in C++ has access to its own address through an important pointer called this pointer.

Uses of this:

1) When local variable's name is same as member's name
2) To return reference to the calling object

When local variable's name is same as member's name

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

```
/* local variable is same as a member's name */
class Test
{
private:
int x;
public:
void setX (int x)
{
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
}
void print() { cout<< "x = " << x <<endl; }
};

int main()
{
Test obj;
int x = 20;
obj.setX(x);
obj.print();
return 0;
}
```

To return reference to the calling object

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

class Test
{
private:
int x;
int y;
public:
Test(int x = 0, int y = 0)
{
this->x = x;
this->y = y;
}
Test &setX(int a)
{
x = a;
return *this;
}
Test &setY(int b)
{
y = b;
return *this;
}
```

```
void print()
 {
cout<< "x = " << x << " y = " << y <<endl;
 }
};

int main()
{
Test obj1(5, 5);

// Chained function calls. All calls modify the same object
// as the same object is returned by reference
obj1.setX(10).setY(20);
obj1.print();
return 0;
}
```

Polymorphism

Polymorphism is derived from two Greek words poly and morph. The word poly means many and morph means form. So polymorphism means many forms.

The process of representing one form in multiple forms is known as Polymorphism.

In C++, polymorphism is divided into two types:

- Compile time polymorphism or Static binding or Early binding : This is achieved by function overloading or operator overloading
- Runtime_polymorphism or Dynamic binding or Late binding : This is achieved by function overriding

A binding refers to the process that is to be used for converting functions and variables into machine language addresses.

Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

In **early binding**, the compiler matches the function call with the correct function definition at compile time. It is also known as static binding or compile time binding.

By default, the compiler goes to the function definition which has been called during compile time.

In **late binding**, the compiler matches the function call with the correct function definition at runtime. It is also known as dynamic binding or runtime binding.

In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.

By default, binding takes place early. Dynamic binding can be achieved by declaring virtual functions.

Virtual functions

In C++, when a derived class inherits from a base class, an object of the derived class may be referred to via a pointer of the base class type instead of the derived class type.

If there are base class methods overridden by the derived class, the method actually called by such a pointer can be bound either early (by the compiler), according to the declared type of the pointer or reference, or late (by the runtime), according to the actual type of the object referred to.

When a base and derived classes have the member functions with the same name, at the time of compilation the compiler does not know which function is to be executed for the function call.

To avoid such problems, user need to use the runtime polymorphism depending on the virtual functions by using the keyword virtual.

The virtual function is a member function that is declared within the base class and redefined in the derived classes.

If the function is **virtual** in the base class, the method is resolved late and the derived class implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer.

If the function is not virtual, the method is resolved early and the function called is selected according to the declared type of the pointer.

The general format of defining virtual functions is:

```
class Base {
        public:
                virtual return-type functionName() {
                        ... // This function is virtual and same function is available in Derived
class
                }
};
class Derived : visibility-mode Base {
        public:
                return-type functionName() {
                        ...
                }
};
```

Let us consider an example:

```
#include <iostream>
using namespace std;
class Base { // This is class Base
public:
        virtual void display() { // This is virtual and tells the compiler that this function binding
        // is done only at runtime
                        cout<< "This is base class display()" <<endl;
                }
};
```

```
class Derived : public Base {  // This is class Derived derived from Base class publicly
        public:
                void display() {
                        cout<< "This is derived class display()" <<endl;
                }
};
int main() {
        Base ob1; // Object is created to the Base class
        Derived ob2; // Object is created to the Derived class
        Base *p; // Pointer object is created to the Base class
        p = &ob1; // Address of Base class object is stored in the pointer object
        p->display(); // Accessing display() method of Base class and it is valid
        p = &ob2; // Address of Derived class object is stored in the pointer object
        p->display(); // Accessing display() method of Derived class and it is also valid
}
```

In the above example, the display() method in Base class is made as virtual.

While accessing display() method through a pointer object it will call the Base class display() or Derived class display() at runtime depending on the address contained in pointer object.

Rules that must be followed when creating virtual functions are:

- Virtual functions should not be static but must be a member of the class.
- They should be defined in public section of the class.
- It is possible to define virtual functions outside the class, in this case, the declaration is done inside the class and the definition is outside the class. The keyword virtual should be placed in declaration part but not in definition part.
- It also possible to return a value from virtual functions.
- Virtual functions may be declared as a friend for another class.
- The prototype in a base class and derived class must be identical for the virtual function to work properly.
- A class cannot have virtual constructors, but can contain a virtual destructor.
- A virtual function is present in the base class but if the same name of the function is not redefined in the derived class then the base class function is invoked every time.
- It is possible to have virtual operator overloading.

Access Private functions using Virtual functions

A user can call private member function of the derived class from the base class pointer with the help of virtual keyword.

The compiler checks for access specifier only at compile time. So at runtime when late binding occurs, it does not check whether it is calling the private function or the public function.

Let us consider an example:

```
#include <iostream>
using namespace std;
class Base { // This is class Base

        public:
                virtual void display() { // This is virtual and tells the compiler that this function
binding is done only at runtime
                        cout<< "This is base class display()" <<endl;
                }
};
class Derived : public Base {  // This is class Derived derived from Base class publicly

        private:
                void display() { // This is a private method
                        cout<< "This is derived class display()" <<endl;
                }
};
int main() {
        Derived ob2; // Object is created to the Derived class
        Base *p; // Pointer object is created to the Base class
        p = &ob2; // Address of Derived class object is stored in the pointer object
        p->display(); // Late binding occurs and which can access the display() method of
Derived class
}
```

In the above example, the display() method in Base class is made as virtual.

The late binding is done at accessing display() method through a pointer object which contains the address of Derived class object. So, it does not check whether the display() is private or public.

Pure Virtual functions

A pure virtual function is a function that has no definition within the base class.

Pure virtual methods typically have a declaration (signature) and no definition (implementation).

A pure virtual function is declared in the base class and cannot be used for any operation.

The class which contains a pure virtual function cannot be used to declare objects, such classes are known as abstract classes.

If anyone attempts to declare an object to abstract classes then the compiler would show an error.

A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract.

All the derived classes without pure virtual functions are called as concrete classesi.e., they can be used to create objects.

The format of a pure virtual function is:

```
virtual return-type function-name() = 0;
```

In C++, a class is abstract if it has at least one pure virtual function.

Let us consider an example:

```
#include <iostream>
using namespace std;
class Base { // This is class Base
        public:
                virtual void display() = 0; // It is a pure virtual function
};
class Derived : public Base {  // This is class Derived derived from Base class publicly
        public:
                void display() { // It is the overridden method of pure virtual function
                        cout<< "This is derived class display()" <<endl;
                }
};
int main() {
        Derived ob; // Object is created to the Derived class
        Base *p; // Pointer object is created to the Base class
        p = &ob; // Address of Derived class object is stored in the pointer
        p->display(); // It will access the display() method of Derived class
}
```

In the above example, the display() method in Base class is made as pure virtual function.

While accessing display() method through a pointer object it will call the Derived class display() at runtime.

Abstract classes

In C++, a class is abstract if it has at least one pure virtual function.

The class which contains a pure virtual function cannot be used to declare objects, such classes are known as abstract classes.

Abstract classes are used to provide an interface for its subclasses. If the derived class does not override the pure virtual function of the base class, then the derived class also becomes an abstract class.

Abstract classes cannot be instantiated but the pointers and references of abstract class type can be created.

An abstract class can have constructors.

```
#include <iostream>
using namespace std;
class Base {
        protected:
                int value;
                public:
                        Base(int number) {
                                value = number;
                        }
};                      virtual void display() = 0;
class Derived : public Base {
        public:
                Derived(int number) : Base(number) {

                }
                void display() {
                        cout<< "The given value : " << value <<endl;
                }
};
int main() {
        Base *p;
        Derived ob(55);
        p = &ob;
        p->display();
}
```

Virtual Destructors

Need of virtual Destructor

A destructor is used to destroy the objects that have been created by a constructor.

A destructor cleans up the storage (memory area of the object) that is no longer accessible.

The compiler automatically invokes the destructor when the object goes out of scope.

Destructors can also be used in inheritance concept. Destructors are invoked in the order opposite to the order in which constructors are called.

While using the polymorphism concept, deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour.

For example, the following program results in undefined behaviour:

```
#include <iostream>
using namespace std;
class Base { // This is class Base

        public:
                Base() {
                        cout<< "This is base class constructor\n";
                }
                ~Base() {
                        cout<< "This is base class destructor\n";
                }
};
class Derived : public Base {  // This is class Derived derived from Base class publicly

        public:
                Derived() {
                        cout<< "This is derived class constructor\n";
                }
                ~Derived() {
                        cout<< "This is derived class destructor\n";
                }

};
int main() {
        Base *p = new Derived; // Base class pointer pointing anonymous object of Derived class
        delete p; // Deleting the memory pointed by p
}
```

In the above example, delete p will only call the Base class destructor, which is undesirable because, the object of Derived class remains unrestricted because its destructor is never called which results in memory leak.

A constructor cannot be virtual because the constructors are always called in an order of base constructor and derived constructor respectively.

So, there is no necessity of making constructor as virtual.

Virtual Destructors

The destructors can be declared as virtual because in polymorphism the correct execution of destructors is only done by using virtual for base class destructors.

Destructors of the base and derived classes are called when a derived class object address pointed by a base class pointer object is deleted.

Let us consider an example:

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

```
class Base { // This is class Base

        public:
                Base() {
                        cout<< "This is base class constructor\n";
                }
                virtual ~Base() {
                        cout<< "This is base class destructor\n";
                }
};
class Derived : public Base {  // This is class Derived derived from Base class publicly

        public:
                Derived() {
                        cout<< "This is derived class constructor\n";
                }
                ~Derived() {
                        cout<< "This is derived class destructor\n";
                }

};
int main() {
        Base *p = new Derived;
        delete p;
}
```

In the above example, *p is a pointer object of the class Base.

The new operator allocates dynamic memory to the class Derived and then the anonymous object address is assigned to the Base class pointer p.

When the memory is allocated to the object of class Derived, it calls the constructors of Base and Derived classes respectively.

While deleting the memory it calls destructors, they are called in the order of Derived class destructors and Base class destructors only by placing virtual at the Base class destructor.

Inline functions

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called.

With inline keyword, the compiler replaces the function call statement with the function code itself and then compiles the entire code.

Thus with inline functions, the compiler does not have to jump to another location to execute the function and then jump back as the code of the called function is already available to the calling program.

If a function is big in terms of executable instructions then compiler can ignore the inline request and treat the function as a normal function.

The syntax for defining the function as inline is:

```
inline return-type function-name(parameters) {
// function code
}
```

```
include <iostream>
using namespace std;
inline int cube(int s) {
return s * s * s;
}

int main() {
int num;
cout<< "Enter a number : ";
cin>> num;
cout<< "The cube of a given number : " << cube(num) << "\n";
}
```

Static Data Members

The data members and member functions of a class may be qualified as static, so there may be static data members and static member functions.

A static data member of a class is just like a global variable for its class, i.e. the static data member is globally available for all the objects of that class type.

The static data members are usually maintained to store values common to the entire class.

For instance, a class may have a static data member keeping track of its number of existing objects.

A static data member has the following properties:

- It is initialized to zero when the first object of its class is created and no other initialization is permitted to the same static data member.
- Only one copy of the static data member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class but its lifetime is the entire program.

The declaration of a static data member within the class definition is similar to any other variable declaration except that it starts with the keyword static as shown below:

Syntax:

```
static data_typemember_name;
```

Example:

```
class Sample {
        static int count;
        ...
        ...
};
```

The above declared static data member count can be defined outside the class as:

Syntax: data_typeclass_name :: member_name =value;

Example:

```
int Sample::count;
```

Note that the type and scope of each static data member must be defined outside the class which is necessary because the static data members are stored separately rather than a part of an object.

Since they are associated with the class rather than an object, they are also known as class variables.

Static variables are initialized to zero by default. A user can also initialize a value to the static variable as:

```
int Sample::count = 10;
```

```
#include <iostream>
using namespace std;
class StaticData {
        char name[25];
        int id;
        static int count;
        public :
                void getData() {
                        cout<< "Enter student id and name : ";
                        cin>> id >> name;
                        count++;
                }
                void getCount() {
                        cout<< "Objects count is : " << count <<endl;
                }
                void putData() {
                        cout<< "Student id : " << id << " name : " << name <<endl;
                }
};
int StaticData::count;
int main() {
        StaticData s1;
        s1.getData();
        s1.getCount();
```

```
    StaticData s2, s3;
    s2.getData();
    s3.getData();
    s1.putData();
    s2.putData();
    s3.putData();
    s3.getCount();
}
```

Static Member Functions

A static member function can access only the static data members of the same class and it does not access any non-static data members.

A static member function can be declared in the class definition by using the keyword static as:

```
static return-type function-name(arguments) {
    ...
}
```

A static member function can be called by using class name instead of object as:

```
class-name::function-name();
```

```
#include <iostream>
using namespace std;
class Item {
        int itemNum;
        float price;
        static int count;
        public :
                void getData() {
                        cout<< "Enter item number and price : ";
                        cin>>itemNum>> price;
                        count++;
                }
                static void getCount() {
                        cout<< "Objects count is : " << count <<endl;
                }
                void putData() {
                        cout<< "Item number : " <<itemNum<< " price : " << price <<endl;
                }
};
int Item::count;
int main() {
        Item i1;
        i1.getData();
        Item::getCount();
        Item i2, i3;
        i2.getData();
        i3.getData();
        i1.putData();
        i2.putData();
```

```
    i3.putData();
    Item::getCount();
}
```

Static Objects

The static keyword can be applied to local variables, member functions, data members and as well as objects in C++.

An object becomes static when the static keyword is used in its declaration.

The static objects are initialized only once and destroyed only when the entire program terminates.

The static objects are active only within their scope but they are alive throughout the program.

In static objects, each data member is initialized to zero by default which does not happen to normal objects.

```
#include <iostream>
using namespace std;
class Sample {
        int num1, num2;
        public :
                void add() {
                        num1 += 5;
                        num2 += 9;
                }
                void show() {
                        cout<< num1 << " " << num2 <<endl;
                }
};
int main() {
        static Sample s;
        cout<< "Before addition : ";
        s.show();
        s.add();
        cout<< "After addition : ";
        s.show();
}
```

UNIT-5

Generic programing with templates and Exceptional handling

Generic programming with

* Template provides generic programming by defining generic classes.

* Template is a technique are passes (or) keyword that allows a using a single function (or) class supports different types of data

* The Goal of generic programming is write the code that is independend of datatype.

* A function that works for all C++ datatypes is called as generic functions.

* Templates associated with classes are called as template classes

Ex:- * If we can create a template for addition of two numbers it helps us to calculate addition of any datatype including int, float, class, double, long, small

* Using template we can create a single function that can process any type of data i.e the formal parameters of functions are template types

original data with some value
actual parameters
formal

* Normally we overload a function when we need to handling different data types this approach increases size of the program and local variables are created in the memory

Defination (or) Declaration of Template:-

To declare a template class following syntax is used

```
template calls <t>
class class_name
{
data member declaration,
member function declaration/definition
};
```

* the first statement in the above syntax also the compiler that the following class is gives template datatype
* t is a variable of template datatype that can be used in following class
* < > it is the angular brackets used to declare template variables.

i) Write a program to show values of different datatypes using constructor overloading and templates

Constructor overloading

```
# include <iostream.h>
# include <conio.h>
class data
{
public:
  data (char c)
  {
  cout << "\n" << "c = " << c << " size in bytes:" << sizeof [c];
  }
  data (int c)
  {
  cout<<"\n"<< "c = "<<c<< "size of in bytes:"sizeof [c];
  }
 data (double)
 {
 cout << "\n" <<"c=" <<c<<"size in bytes:"<< c sizeof [c];
 }
};
```

```
void main()
{
clrscr();
data h('A');
data i(100);
data j(68.2);
getch();
}
```

O/P

C = A
Size in bytes = 1
C = 100
Size in bytes = 2
C = 68.2
Size in bytes = 8

Template:-

```
# include<iostream.h>
#l include<conio.h>
Template <class T>
Class data
{
 public;
  data (T c)
 {
cout <<"\n"<<"c="<<c<<" Size in bytes:"<<sizeof (c);
 }
};
void main()
{
 clrscr();
 data <char> h('A');
 data <int> i(100);
 data <float> j (3.12);
 getch();
}
```

O/P

C=A
Size in bytes=1
C=100
Size in bytes =2
C=68.2
Size in bytes =8

Normal function Templates (or) Function templates:-

* A normal function can also used template arguments.
* The difference between normal and member function is normal member function (or) defined without classes that is they are not members of any class member functions are defined by using classes that is they can be invoked by using object and DOT operator.

write a program to define a normal template function

```
#include<iostream.h>
#include<conio.h>
 template <class T>
   void show(T x)
   {
    cout<<"x value:"<<x;
   }
   void main()
    {
      clrscr();
       int i=100;
       char c='A';
       float f=68.50;
       show(i);
       show(c);
       show(f);
       getch();
       }
```

output:-

x value=100
x value= A
x value=68.50

2) write a program to define data members of template type.

```
#include <iostream.h>
#include <conio.h>
template <class t>
class data
{
public:
  data(t x)
  {
   cout <<"x value:" <<x;
  }
};
void main()
{
 clrscr();
 data <int> i(100);
 data <char> c(A);
 data <float> f(68.50);
 getch();
}
```

Output

x value =100;

x value = A;

x value = 68.50;

write a program find out square of the given value by using template

```
#include<iostream.h>
#include<conio.h>
template<class T>
class square
{
public:
    square(T x)
    {
    cout<<"x value:"<<x*x;
    }
};
void main()
{
clrscr();
square<int>i(10);
square<float>f(6.5);
getch();
}
```

2) write a c++ program for swapping two values using function templates

```
main()
{
int a,b;
cout<<" enter a,b values:";
cin>>a>>b;
a=a+b;
b=a-b;
a=a-b;
cout<<" after swapping "<<a<<b;
}
```

without using templates

```
#include<iostream.h>
void swap(int *a, int *b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
}
void main()
{
 int x,y;
 clrscr();
 cout<<" enter any two integers";
 cin>>x>>y;
 cout<<"Before swapping two numbers are:"<<x<<y;
  swap(&x, &y);
 cout<<"after swapping two numbers are:"<<x<<y;
 getch();
}
```

Using template :-

```
#include <iostream.h>
#include <conio.h>
template <class T>
void swap(T *a, T *b)
{
 T temp;
 temp = a;
 a = b;
  b = temp;
 }
void main()
{
  int x,y
  float c,d;
  clrscr();
   cout <<" Enter any two integers values";
   cin >>x>>y
cout<<" Before swaping two numbers are ="<<x<<y;
  swap (&x, &y);
cout<<" after swaping two numbers are ="<<x<<y;
 cout<< "Enter any two float values";
  cin>>c>>d;
 cout<<" Before swapping two float values are"<<c<<d;
  swap(&c,&d)
 cout <<" after swapping two float values are"<< c<<d;
  getch();
 }
```

2 Write a program to find out maximum of two numbers by using templates.

```
#include <iostream.h>
#include <conio.h>
template <class T>
void max(T a, T b)
{
  if (a>b)
  cout << "a is maximum number";
  else
  cout << "b is maximum number";
}
void main()
{
  int a,b;
  float c,d;
  clrscr();
  cout << "read a,b integer values";
  cin >> a >> b;
    max(a,b);
  cout << "read c,d float values";
  cin >> c >> d;
    max(c,d);
  getch();
}
```

Output:- Read a,b Integer values 10 12

b is maximum number

Read c,d float values 12.50 10.50

a is maximum number

write a c++ program to illustrate -template class

```
#include <iostream.h>
#include <conio.h>
template <class T1,T2>
  class test
  {
  public:
  T1 a;
  T2 b;
  test (T1 x, T2 y)
   {
     a=x;
     b=y;
   }
   void show( )
   {
   cout <<"a=" <<a<<"\n";
   cout <<"b=" <<b;
   }
};
void main( )
{
 clrscr( );
 test <int, float> t1(10, 20.50);
 test <float, int> t2(20.50,10);
  t1.show( );
  t2.show( );
  getch( );
}
```

output:

a = 10

b = 20.50

a = 20.50

b = 10

Overloading of template functions:-

* A template function supports overloading mechanism it can be overloaded by normal function (or) template function.
* The compiler follows the following rules for choosing appropriate function when program consist of template and normal function

i) search for accurate match of functions if found it is invoked

ii) In case no match is found on error will be reported

Write a program to overload template function

```
#include <iostream.h>
#include <conio.h>
template <class t>
void show(t a)
{
 cout <<" template variable =" << a;
}
void show(int a)
{
cout <<" integer variable =" <<a;
}
void main()
{
 clrscr();
 show('A');
 show(100);
 show(10.50);
 getch();
}
```

O/P:-

template variable = A

Integer variable : 100

template variable = 10.50

Bubble sort using Function template:-

* The process of arranging the giving elements either in ascending (or) descending order is called as sorting.

* Bubble sort is one of the technique for sorting given elements

* In bubble sort we are comparing adjacent elements

ex:- apply bubble sort technique for following data and place the data in ascending order.

12, 3, 0, -3, 1, -9

* place the given elements in array

a[5] =

0	1	2	3	4	5
12	3	0	-3	1	-9

Step Step: 1

Compare a[0] and a[1]

Compare 12 and 3

12>3

Swapping two elements

0	1	2	3	4	5
3	12	0	-3	1	-9

Compare a[1] and a[2]

Compare 12 and 0

12>0

Swapping two elements

0	1	2	3	4	5
3	0	12	-3	1	-9

Compare a[2] and a[3]

Compare 12 and -3

12>-3

Swapping two elem

0	1	2	3	4	5
3	0	-3	12	1	-9

Compare a[3] and a[4]

Compare 12 and 1

12>1

Swapping two elements

0	1	2	3	4	5
3	0	-3	1	12	-9

Compare a[4] and a[5]

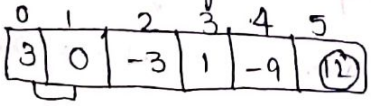
Compare 12 and -9

12>-9

0	1	2	3	4	5
3	0	-3	1	-9	12

* Therefore in step one we placing (or) sorting the largest element in given array.

Step:2 a[5] =



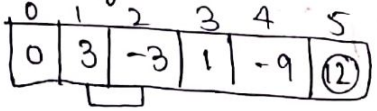
Step:2

Compare a[0] and a[1]

Compare 3 and 0

3>0

Swapping two elements

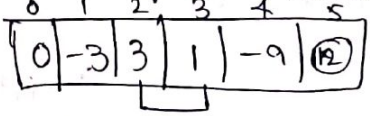


Compare a[1] and a[2]

Compare 3 and -3

3 > -3

Swapping two elements

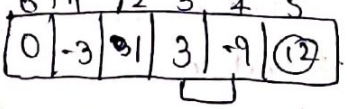


Compare a[2] and a[3]

Compare 3 and 1.

3>1

Swapping two elements

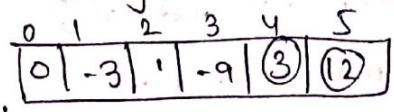


Compare a[3] and a[4]

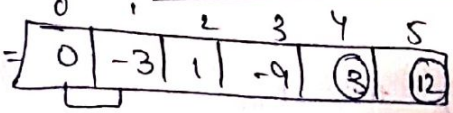
Compare 3 and -9

3>-9

Swapping two elements



a[5] =

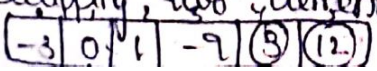


Step:3:- Compare a[0] and a[1]

Compare 0 and -3

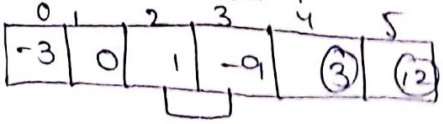
0>-3

Swapping two elements



Compare a[1] and a[2]

Compare 0 and 1

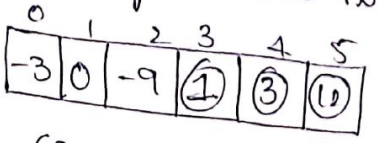


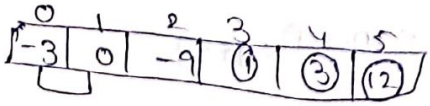
Compare a[2] and a[3]

compare 1 and -9

1 > -9

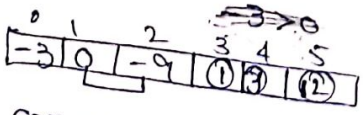
Swapping of two numbers



Now a[5] = 

Step:4 Compare a[0] and a[1]

Compare -3 and 0.

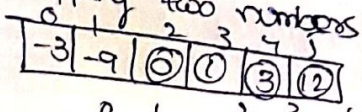


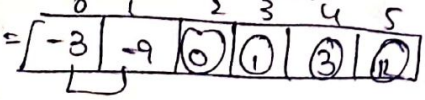
Compare a[1] and a[2]

Compare 0 and -9

0 > -9

Swapping two numbers



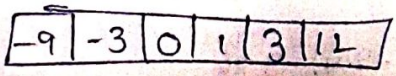
a[5] = 

Step:5 Compare a[0] and a[1]

Compare 0 and 1

-3 > -9

Swapping two elements



**

Write a program sort the given elements by using bubble, using template function.

```
#include<iostream.h>
#include<conio.h>
template<class T>
void Bsort(T a[], int n)
{
 for(int i=0; i<n-1; i++)
  {
  for(int j=0; j<(n-i-1); j++)
   {
     if(a[j]>a[j+1])
     {
       T temp;
      temp=a[j];
       a[j]=a[j+1];
      a[j+1]=temp;
     }
   }
  }
}
void main()
{
 int i;
 clrscr();
 int a[5]={12,0,3,1,-9,-3}
 float b[5]={12.5,0.4,3.2,1.6,-9.4,-3.8}
  Bsort(a,6); - function calling statement
  Bsort(b,6);
 cout<<" after sorting integer elements are";
   for(i=0; i<6; i++)
   cout<<a[i]<<"\t";
```

```
cout <<"after sorting float elements";
for (i=0; i<6; i++)
  cout <<b[i]<<"\t";
}getch();
```

Output :- after sorting integer elements are

-9, -3, 0, 1, 3, 12

after sorting float elements are

-9.4, -3.8, 0.4, 1.6, 3.2, 12.5

Difference between Template and Macros :- (preprocessing)

* Macro's are preprocessing statements and not type safe that is a macro defined for integer operation can't accept the float data.
* It is difficult to find out errors in macros.

Write a program to perform increment operation by using Macros and templates.

```
#include <iostream.h>
#include <conio.h>
#define mac(x) ++x
void main()
{
 int a,c;
 clrscr();
 a=10;
 c=mac(a);
 cout<<"after incrementation of a="<<c;
 getch();
}
```

```
#include <iostream.h>
#include <conio.h>
template <class T>
T mac(T x);
{
 ++x;
 return x;
}
void main()
{
 int a,c;
 clrscr();
 a=10;
 c=mac(a);
 cout<<"after incrementation of
 getch();          a="<<c;
}
```

Exceptional handling:-

* Developing an error free program is the objective of programer. the programer have to take care to prevent errors. Errors can be handled by using exception handling Mechanism.

* the errors may be syntax errors (or) logical errors

* the logical errors occured in the program due to poor understanding of the program (logic)

* Syntax errors occured dee to in the program due to poor understanding of the programing languages

* C++ Provides exceptional handling procedure to reduce the errors that a programer makes

* programer always faces unusual errors while writting programs

* An exception is abnormal termination of a program which is executed in a program at run time.

Principles of exception Handling:-

* the goal of exception Handling is to create a routine that identifies and stops an exceptionals inorder to execute program properly.

* the routine needs to carry the following responsibilities

i) Identify an exception (or) Identify an problem

ii) Warn that an error has come

iii) Accept the error message

iv) Handling the error message

v) An exception is an object in his shift from path of the program where an error occurs to that path of

the program and which is going to handle the error.

Keywords of exceptional handling:-

* Exception handling technique passes control of the program from a location of exception in a program to an exception handler with the try block.

i) Try Block:- * the try keyword is followed by a series of statements enclosed with curly braces.

* Try keyword is used to identifying an exceptions

ii) throw keyword (or) Block:- * the function of throw statements are to send the exception found.

* the declaration of throw statement is as given below.

```
throw (exception);
  (or)
throw exception;
```

* The argument exception is allowed in any type of data and it may be a constant value also

iii) catch Block (or) keyword:- * like try blocks catch block also contains a series of statements enclosed within curly brass.

* catch block associated with try block catches the exceptions thrown and the control is transfered from try block to catch block.

Syntax:-

```
try
{
  Statement 1;
  Statement 2;
  :
  :
  statement n;
}
catch(type z argument)
{
 Statement 1;
 :
 statement n; }
```

1) write a program to identify exceptions by using exception handling mechanism.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a,b,x;
clrscr();
cout<<"Enter any two numbers";
cin>>a>>b;
x=a-b;
try
{
if(x!=0)
cout<<"Result="<<b/x;
else
throw(x);
}
}
catch(int i)
{
cout<<"exception caught:"<<i;
}
```

output:-

Enter any two numbers 20 20

Result = 1

Enter any two numbers 10 10

exception caught : 0

Multiple Catch statements:- It is possible to a program segment has more than one catch statement with a single try block.

Syntax:-

```
try
{
 // try block
}
catch (Type 1 argument)
{
  // catch Block
}
Catch (Type 2 argument)
{
 // catch block
 }
 :
Catch (type N argument)
{
 //catch Block
}
```

* when an exception is thrown the exception handler are searched in order for an appropriate match

write a program to catch multiple exceptions.

```
#include <iostream.h>
#include <conio.h>
void num(int k)
{
 try
 {
 if(k==0) throw k;
  else
   if(k>0) throw 'p';
  else
   if (k<0) throw '0';
  cout<<"*** try block ***\n";
 }
  catch (int i)
```

```
{
cout<<"\n caught an exception \n";
}
}catch(char ch)
void main(-)
{
cout<<"\n caught an exception \n"<<ch;
}
}

void main()
{
 num(0);
 num(5);
 num(-1);
 getch();
}
```

output:- caugh an exception: 0

caught

catch in multiple exceptions:-

*It is also possible to define a single catch block from one (or) more exceptions of different datatypes. In such situation single catch block is used the catch exceptions thrown by multiple throw statements

Syntax:-

```
catch(.....)
{
  //catch block statements
}
```

write a program to handle multiple exceptions by using a single catch block.

```
# include <iostream.h>
# include <conio.h>
void num(int k)
{
 try
 {
```

```
    if(k==0) throw k;
    else
    if(k>0)throw 'p';
    else
     if (k<0) throw 0.5;
    }
}
catch (....)
 {
 cout<<"exception caught";
 }
void main()
 {
  clscr();
  num(0);
  num(-1);
  num(1);
  getch();
 }
```

advantage :- reduce the size of the program.

disadvantage :- we don't find what kind of exception

output:-

exception caught

exception caught

exception caught

Specifying exceptions:-

The specified exceptions are used when we want to tell bind the function to throw only specified exceptions using a throw list condition can do this.

Syntax:-

```
Returntype functionname(parameters list) throw (type of exception)
 {
   function Body
 }
```

write a program to handle only integer type of exceptions.

```
#include<iostream.h>
#include <conio.h>
void num(int k) throw(int)
 {
  try
    {
      if(k==0) throw k;
      else if(k<0) throw k;
      else
         throw 0.5;
    }
  }
 catch (int i)
 {
  cout<<"Integer exception caught"<<i;
 }
 catch (char ch)
 {
  cout<<" character exception caught"<<ch;
 }
 catch(float f)
 {
 cout<<"float exception caught"<< f;
 }
 void main()
 {
  clrscr();
   num(0);
   num(-1);
  num(1);
   getch();
 }
```

output:-

Integer exception caught 0

C and C++

7. Exceptions — Templates

Stephen Clark

University of Cambridge
(heavily based on last year's notes (Andrew Moore) with thanks to Alastair R. Beresford and Bjarne Stroustrup)

Michaelmas Term 2011

Exceptions

- Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it
- C++ provides exceptions to allow an error to be communicated
- In C++ terminology, one portion of code throws an exception; another portion catches it.
- If an exception is thrown, the call stack is unwound until a function is found which catches the exception
- If an exception is not caught, the program terminates

Throwing exceptions

- Exceptions in C++ are just normal values, matched by type
- A class is often used to define a particular error type:
  `class MyError {};`
- An instance of this can then be thrown, caught and possibly re-thrown:

```
1 void f() { ... throw MyError(); ... }
2 ...
3 try {
4   f();
5 }
6 catch (MyError) {
7   //handle error
8   throw; //re-throw error
9 }
```


Conveying information

- The "thrown" type can carry information:

```
1  struct MyError {
2    int errorcode;
3    MyError(i):errorcode(i) {}
4  };
5
6  void f() { ... throw MyError(5); ... }
7
8  try {
9    f();
10 }
11 catch (MyError x) {
12   //handle error (x.errorcode has the value 5)
13   ...
14 }
```

Handling multiple errors

- Multiple catch blocks can be used to catch different errors:

```
try {
   ...
}
catch (MyError x) {
   //handle MyError
}
catch (YourError x) {
   //handle YourError
}
```

- Every exception will be caught with `catch(...)`
- Class hierarchies can be used to express exceptions:


```cpp
#include <iostream>

struct SomeError {virtual void print() = 0;};
struct ThisError : public SomeError {
  virtual void print() {
    std::cout << "This Error" << std::endl;
  }
};
struct ThatError : public SomeError {
  virtual void print() {
    std::cout << "That Error" << std::endl;
  }
};
int main() {
  try { throw ThisError(); }
  catch (SomeError& e) { //reference, not value
    e.print();
  }
  return 0;
}
```


Exceptions and local variables

- When an exception is thrown, the stack is unwound
- The destructors of any local variables are called as this process continues
- Therefore it is good C++ design practise to wrap any locks, open file handles, heap memory etc., inside a stack-allocated class to ensure that the resources are released correctly

Templates

- Templates support meta-programming, where code can be evaluated at compile-time rather than run-time
- Templates support generic programming by allowing types to be parameters in a program
- Generic programming means we can write one set of algorithms and one set of data structures to work with objects of any type
- We can achieve some of this flexibility in C, by casting everything to `void *` (e.g. `sort` routine presented earlier)
- The C++ Standard Template Library (STL) makes extensive use of templates

An example: a stack

- The stack data structure is a useful data abstraction concept for objects of many different types
- In one program, we might like to store a stack of `int`s
- In another, a stack of `NetworkHeader` objects
- Templates allow us to write a single generic stack implementation for an unspecified type `T`
- What functionality would we like a stack to have?
  - `bool isEmpty();`
  - `void push(T item);`
  - `T pop();`
  - ...
- Many of these operations depend on the type `T`

Creating a stack template

- A class template is defined as:

```
1 template<class T> class Stack {
2   ...
3 }
```

- Where `class T` can be any C++ type (e.g. `int`)
- When we wish to create an instance of a `Stack` (say to store `int`s) then we must specify the type of `T` in the declaration and definition of the object: `Stack<int> intstack;`
- We can then use the object as normal: `intstack.push(3);`
- So, how do we implement `Stack`?
  - Write `T` whenever you would normally use a concrete type


```
template<class T> class Stack {

  struct Item { //class with all public members
    T val;
    Item* next;
    Item(T v) : val(v), next(0) {}
  };

  Item* head;

  Stack(const Stack& s) {}              //private
  Stack& operator=(const Stack& s) {} //

public:
  Stack() : head(0) {}
  ~Stack();
  T pop();
  void push(T val);
  void append(T val);
};
```



```
1  #include "example16.hh"
2
3  template<class T> void Stack<T>::append(T val) {
4    Item **pp = &head;
5    while(*pp) {pp = &((*pp)->next);}
6    *pp = new Item(val);
7  }
8
9  //Complete these as an exercise
10 template<class T> void Stack<T>::push(T) {/* ... */}
11 template<class T> T Stack<T>::pop() {/* ... */}
12 template<class T> Stack<T>::~Stack() {/* ... */}
13
14 int main() {
15   Stack<char> s;
16   s.push('a'), s.append('b'), s.pop();
17 }
```


Template details

- A template parameter can take an integer value instead of a type:
  `template<int i> class Buf { int b[i]; ... };`
- A template can take several parameters:
  `template<class T,int i> class Buf { T b[i]; ... };`
- A template can even use one template parameter in the definition of a subsequent parameter:
  `template<class T, T val> class A { ... };`
- A templated class is not type checked until the template is instantiated:
  `template<class T> class B {const static T a=3;};`
  - `B<int> b;` is fine, but what about `B<B<int> > bi;`?
- Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object

Default parameters

- Template parameters may be given default values

```
1 template <class T,int i=128> struct Buffer{
2   T buf[i];
3 };
4
5 int main() {
6   Buffer<int> B; //i=128
7   Buffer<int,256> C;
8 }
```


Specialization

- The `class T` template parameter will accept any type `T`
- We can define a specialization for a particular type as well:

```
1  #include <iostream>
2  class A {};
3
4  template<class T> struct B {
5    void print() { std::cout << "General" << std::endl;}
6  };
7  template<> struct B<A> {
8    void print() { std::cout << "Special" << std::endl;}
9  };
10
11 int  main() {
12   B<A> b1;
13   B<int> b2;
14   b1.print(); //Special
15   b2.print(); //General
16 }
```


Templated functions

- A function definition can also be specified as a template; for example:

```
1 template<class T> void sort(T a[],
2                              const unsigned int& len);
```

- The type of the template is inferred from the argument types:
  `int a[] = {2,1,3}; sort(a,3);` $\Longrightarrow$ `T` is an `int`
- The type can also be expressed explicitly:
  `sort<int>(a)`
- There is no such type inference for templated classes
- Using templates in this way enables:
  - better type checking than using `void *`
  - potentially faster code (no function pointers)
  - larger binaries if `sort()` is used with data of many different types


```cpp
#include <iostream>

template<class T> void sort(T a[], const unsigned int& len) {
  T tmp;
  for(unsigned int i=0;i<len-1;i++)
    for(unsigned int j=0;j<len-1-i;j++)
      if (a[j] > a[j+1]) //type T must support "operator>"
       tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
}

int main() {
  const unsigned int len = 5;
  int a[len] = {1,4,3,2,5};
  float f[len] = {3.14,2.72,2.54,1.62,1.41};

  sort(a,len), sort(f,len);
  for(unsigned int i=0; i<len; i++)
    std::cout << a[i] << "\t" << f[i] << std::endl;
}
```


Overloading templated functions

- Templated functions can be overloaded with templated and non-templated functions
- Resolving an overloaded function call uses the "most specialised" function call
- If this is ambiguous, then an error is given, and the programmer must fix by:
  - being explicit with template parameters (e.g. `sort<int>(...)`)
  - re-writing definitions of overloaded functions
- Overloading templated functions enables meta-programming:

Meta-programming example

```
1  #include <iostream>
2
3  template<unsigned int N> inline long long int fact() {
4    return N*fact<N-1>();
5  }
6
7  template<> inline long long int fact<0>() {
8    return 1;
9  }
10
11 int main() {
12   std::cout << fact<20>() << std::endl;
13 }
```


Exercises

1. Provide an implementation for:
   `template<class T> T Stack<T>::pop();` and
   `template<class T> Stack<T>::~Stack();`
2. Provide an implementation for:
   `Stack(const Stack& s);` and
   `Stack& operator=(const Stack& s);`
3. Using meta programming, write a templated class `prime`, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.
4. How can you be sure that your implementation of class `prime` has been evaluated at compile time?

