OBJECT ORIENTED POGRAMMING WITH C++

1. What is type conversion?

Type Conversion is the process of converting one predefined type into another type. When variables of one type are mixed with variables of another type, a type conversion will occur. C++ facilitates the type conversion into the following two forms:

Implicit Type Conversion: An implicit type conversion is a conversion performed by the compiler without programmer's intervention. It is applied whenever different data types are intermixed in an expression, so as not to lose information

```
short x=6000;
int y;
y=x; // data type short variable x is converted to int and is assigned to the integer variable y.
```

Explicit Type Conversion: The explicit conversion of an operand to a specific type is called type casting. An explicit type conversion is user-defined that forces an expression to be of specific type.

Syntax: (type) expression

```
#include <iostream.h>
void main( )
{
int a;
float b, c;
cout << "Enter the value of a:";
cin >> a;
cout << "Enter the value of b:";
cin >> b;
c = float(a)+b;
cout << "The value of c is:" << c;
getch();
}
```

In the above program “a” is declared as integer and “b” and “c” are declared as float. In the type conversion statement namely **c = float (a) +b;** The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable c with value as 22.5

2. Why C++ is called OOP language?

C++ is called object oriented programming (OOP) language because C++ language views a problem in terms of objects involved rather than the procedure for doing it. An object is an identifiable entity with some characteristics and behaviors. While programming using object oriented approach, the characteristics of an object are represented by its data and its behavior is represented by its functions/operators associated. Therefore, in object oriented programming object represent an entity that can store data and has its interface through functions.

Advantages

Advantages of OOP Programming are that it makes the program less complex, enhances readability. Components are reusable and extendible. Object oriented programming is always good for writing large business logics and large applications or games. Several components of a large program can be easily extended by introducing new classes or introducing new function/operators. OOPs are also very much desired for maintenance and long term support. C++ supports Encapsulation, Data hiding, inheritance and polymorphism so that C++ is a OOP Language.

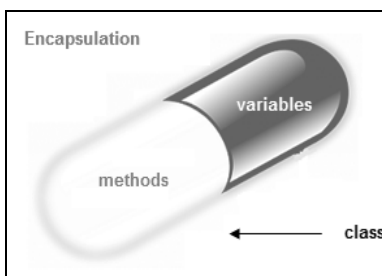
3. **Differentiate between C and C++?**

C	C++
C was developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979 with C++'s predecessor "C with Classes".
When compared to C++, C is a subset of C++.	C++ is a superset of C. C++ can run most of C code while C cannot run C++ code.
C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language.
C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance.	Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance.
In C (because it is a procedural programming language), data and functions are separate and free entities.	In C++ (when it is used as object oriented programming language), data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of

	the object.
In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.

4. What is encapsulation?

Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class. It is one of the popular features of Object Oriented Programming (OOPs) that helps in data hiding.



Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.

Example:

```
class   sum
{
private: int a,b,c;
public:
void add ( )
     {
cout<<"enter any two numbers: ";
cin>>a>>b;
c=a+b;
cout<<"sum: "<<c;
      }
```

5. What is data hiding?

Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or

intended changes. Data hiding also reduces system complexity for increased robustness by limiting interdependencies between software components. Data hiding is also known as data encapsulation or information hiding.

6. **What are the different features of C++?**

C++ is object oriented programming language and it is a very simple and easy language

Important Features of C++

1. Simple
2. Portability
3. Powerful
4. Platform dependent
5. Object oriented
6. Case sensitive
7. Compiler based
8. Syntax based language
9. Use of Pointers

7. **What is Garbage collection**?

Garbage collection is the systematic recovery of pooled computer storage that is being used by a program when that program no longer needs the storage. This frees the storage for use by other programs (or processes within a program). In older programming languages, such as C and C++, allocating and freeing memory is done manually by the programmer. A GC has two goals: any unused memory should be freed, and no memory should be freed unless the program will not use it anymore. In C++, Destructors and delete operator are used for garbage collection.

8. **What is DMA(Dynamic Memory Allocation)**

There are two ways to allocate memory for storing data. They are

1. Compile time allocation or Static memory allocation
2. Run time memory allocation or Dynamic memory allocation

In Dynamic memory allocation, Programmers can dynamically allocate storage space while the program is running, but programmers cannot create new variable names "on the fly", and for this reason, dynamic allocation requires two criteria:

- ➢ Creating the dynamic space in memory
- ➢ Storing its address in a pointer (so that space can be accessed)

The operator “new” is used to allocate the memory dynamically. Memory de-allocation is also a part of this concept where the "clean-up" of space is done for variables or other data storage. It is

the job of the programmer to de-allocate dynamically created space. For de-allocating dynamic memory, we use the delete operator. In other words, dynamic memory Allocation refers to performing memory management for dynamic memory allocation manually.
Memory in your C++ program is divided into two parts:
stack: All variables declared inside any function takes up memory from the stack.
heap: It is the unused memory of the program and can be used to dynamically allocate the memory at runtime.
To allocate space dynamically, use the unary operator new, followed by the type being allocated.

```
int * p; // declares a pointer p
p = new int;  // dynamically allocate an int for loading the address in p
double * d;  // declares a pointer d
d = new double;  // dynamically allocate a double and loading the address in p
```

When programmers think that the dynamically allocated variable is not required anymore, they can use the delete operator to free up memory space.

```
delete var_name;
```

Example:

```
delete p;
```

9. **What is a class? Give the syntax**.

Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of circle. There may be many circles with different measurements but all of them will share a common property like all of them will have radius. So here, circle is the class and radius is its property. Circle 1, circle 2 etc. are its objects.

Syntax:

```
class class_name
 {
                    access_specifier_1:
                    member1;
      access_specifier_2:
                     member2;
  ...
} object_names;
```

Example:

```
class circle
{
        double radius; // data member
        public:
        double area( ) // member function
        {
        return 3.14*radius*radius;
        }
}circle1;
```

10. What is an object? How do you create an object?

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax for creating object:

```
ClassName ObjectName;
```

Ex: circle circle1;

11. What is function overloading?

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters. Function overloading can be considered as an example of polymorphism feature in C++.

```
#include <iostream.h>

void display(int);
void display(float);
void display(int, float);

void main( )
 {
```

```
    int a = 5;
    float b = 5.5;

    display(a);
    display(b);
    display(a, b);
    getch( );
}

void display(int x) {
    cout << "Integer number: " << x << endl;
}

void display(float y) {
    cout << "Float number: " << y << endl;
}

void display(int m, float n) {
    cout << "Integer number: " << m;
    cout << " and float number:" << n;
}
```

Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5

12. What is a constructor?

A constructor is a special function which is used to initialize the objects. It has same name as the class itself. Constructors don't have return type. A constructor is automatically called when an object is created.

There are 3 types of constructors
1) Default constructor
2) Parameterized constructor
3) Copy constructor

Example program

```
Class  construct
 {
Public:
    int a, b;
    construct( )    // Default Constructor
```

```
    {
        a = 10;
        b = 20;
    }
};
void main( )
{
        construct   c;
          cout << "a: "<< c.a << endl << "b: "<< c.b;
          getch( );
 }
```

Output:
a=10
b=20

13. What is operator over loading?

Generally operators are used on fundamental data types. Operator overloading is an important concept in C++, where operators can be used on user defined data types also. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.
Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.
. (Dot)   ::(scope resolution)   *(pointer selector)   ?:(Ternary)   sizeof(size operator)
There are two types of operator overloading:
1) Unary operator over loading (works on single operand)
2) Binary operator over loading (works on two operands)
Example:

```
#include<iostream.h>
#include<conio.h>
class test
{
int a;
public:
test( )
{
a=0;
}
```

```
void operator++( )
{
a++;
}
void operator- -( )
{
a - -;
}
void show( )
{
cout<<"a="<<a<<endl;
};
void main( )
{
test t;
clrscr( );
t++;
t.show( );
t- -;
t.show( );
getch( );
}
```

Output:

a=1

a=0

14. What are visibility modes in C++?

Visibility Modes: Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

Syntax: **Class subclass: public superclass**

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

Syntax: **Class subclass: superclass** (by default it is private)

3) Protected Inheritance

In protected mode, the public and protected members of Super class become protected members of Sub class.

Syntax: **Class subclass: protected superclass**

15. What are new and delete operators?

Dynamic memory allocation in C++ refers to performing memory allocation manually by programmer. This can be done with new and delete operators.
Syntax to use new operator: **pointer-variable = new data-type;**
Example: **int *p;**
p= new int(10);
To deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.
Syntax: **delete pointer-variable;**
Example: **delete p;**

16. What is inheritance?

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the Parent or Base or Super class. And, the class which inherits properties of other class is called Child or Derived or Sub class. Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

Types of inheritance in C++

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Syntax:

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

17. What is polymorphism?

The word polymorphism means ability to have many forms. Polymorphism means that some code or operations or objects behave differently in different contexts.
Polymorphism is a feature of OOPs that allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:
1) Compile time Polymorphism – This is also known as static (or early) binding.
2) Runtime Polymorphism – This is also known as dynamic (or late) binding.

18. What is function overriding?

If a function is defined in both base class and derived class with same name, same parameters or signature, then that function is said to be overridden and the process is called function overriding or method overriding. In function overriding, signature (Proto type) of base class function and derived class must be same.
Signature (Function proto type) includes: a) Number of arguments b) Type of arguments c) Sequence of arguments
In below example same method "show ( )" is present in both base and derived class with same name and signature.

Example of Method Overriding in C++

```
#include<iostream.h>
#include<conio.h>
Class    Base
{
 public:
 void show( )
  {
  cout<<"Base class";
  }
};
class    Derived:   public Base
{
 public:
 void show( )
  {
  cout<<"Derived Class";
  }
};

void main()
{
```

```
Base b;            //Base class object
Derived d;         //Derived class object
b.show( );
d. show( );
getch( );
}
```

Output
Base class
Derived Class

19. What is an abstract class?

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. We cannot declare an object of an abstract class.

Example:

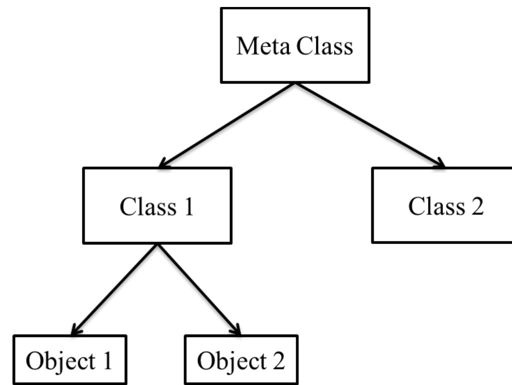
```
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show( ) = 0;

};
```

In the above example, class Test is called abstract class because it contains a pure virtual function show( ). Now we cannot create an object for this class.

20. What is a Meta class?

In object-oriented programming, a Meta class is a class whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a meta class defines the behavior of certain classes and their instances.



21. What is a virtual function?

A virtual function a member function, which is declared in base class and is re-defined (Overridden) by derived class. When you refer to a derived class object using a pointer or a reference to the base class you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.

Example

```
#include<iostream.h>
class   base
{
public:
            virtual void show( )
{
 cout<<" in base \n";
}
};
  class derived: public base
{
public:
   void show( )
{ cout<<"in derived \n";
}
};
void main( )
{
base *b;
```

```
derived d;
b=&d;
b->show( );     // run-time polymorphism
getch( );
}
```

OUTPUT: In Derived

22. What is a pure virtual function?

A pure virtual function (or abstract function) in C++ is a virtual function for which don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

```
// An abstract class
class   Test
{
        // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

  /* Other members */
};
```

23. What is dynamic binding?

A virtual function is declared by using the 'virtual' prefix to the declaration of that function. A call to virtual function is resolved at the run-time by analyzing the actual type of object that is calling the function. This is 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, early binding takes place. So if by any means we tell the compiler to perform late binding, then the problem in the previous example can be solved.

Example

```
#include<iostream.h>
class   base
{
public:
            virtual void show( )
{
 cout<<" in base \n";
}
};
  class derived: public base
```

```
{
public:
  void show( )
{ cout<<"in derived \n";
}
};
void main( )
{
base *b;
derived d;
b=&d;
b->show( );     // Dynamic binding occurs
getch( );
}
```

OUTPUT: In Derived

24. What is an exception handling?

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

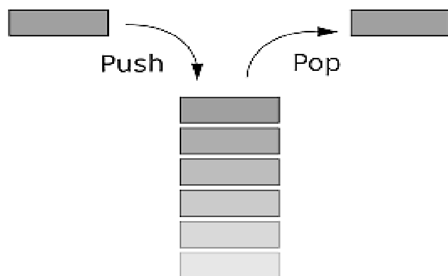
```
try
{
  //  code
 }
catch( ExceptionName e1 )
{
  // catch block
}
```

```
catch( ExceptionName e2 )
{
  // catch block
}
catch( ExceptionName eN )
{
  // catch block
}
```

We can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

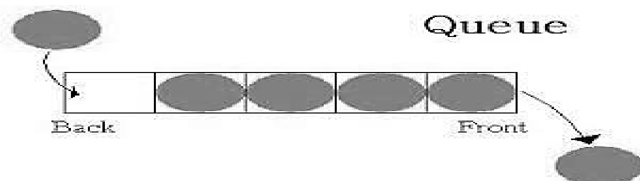
25. Explain Stack, Queue and Linked list

Stacks are a type of container adaptors with LIFO (Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.



The functions associated with stack are:
empty() – Returns whether the stack is empty
size() – Returns the size of the stack
top() – Returns a reference to the top most element of the stack
push(g) – Adds the element 'g' at the top of the stack
pop() – Deletes the top most element of the stack

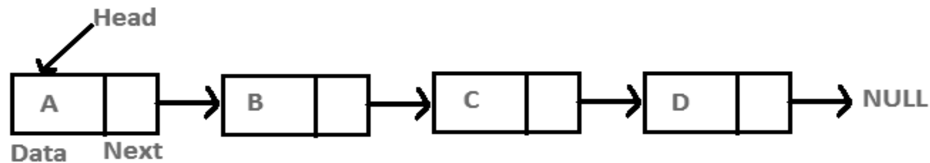
Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.



The functions supported by queue are :
empty() – Returns whether the queue is empty
size() – Returns the size of the queue
front() – Returns a reference to the first element of the queue
back() – Returns a reference to the last element of the queue

push(g) – Adds the element 'g' at the end of the queue
pop() – Deletes the first element of the queue.

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



Why Linked List?
Arrays can be used to store linear data of similar types, but arrays have following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements have to shifted.
For example, in a system if we maintain a sorted list of IDs in an array id[ ].
id[ ] = [1000, 1010, 1050, 2000, 2040].
There are three common types of Linked List.
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

ESSAY TYPE QUESTIONS

Unit-I

1. Differences between Structured Vs Object Oriented Programming

2. What are the Data Types in C++?

3. Explain structure of C++ program

4. Explain about control Structures in c++

5. Explain Operators in C++

Unit-II

1. What is a class? What is the structure of a class?

2. Explain the following i) Object ii) Member function iii) Message passing

3. Explain Dynamic memory allocation

4. What is a constructor? Explain different types of constructors with examples.

Unit-III

1. Explain about Access specifiers (visibility modes) in C++ with suitable examples

2. Explain about function overloading with example

3. Explain unary and binary operator overloading with an example

4. What is inheritance? Explain single, multiple and multi- level inheritance with example

Unit-IV

1. Explain method overriding with example

2. Explain about virtual function with example

3. What is pure virtual function explain.

4. Explain about stream classes

Unit-V

1. Explain exception Handling Mechanism in C++ with example

2. Explain about stack and its applications?

3. Explain about types of Linked lists.

4. Explain about types Queues and its applications.

1. What is a class and object? Explain with example

Class: The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of "employee". There may be many employees but all of them will share some common properties like all of them will have names, department, salary etc. So here, Employee is the class and Id, name, salary are their properties. Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

Object: An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Syntax for creating the class :

```
class Class_Name
 {
   data member;
   method;
 }
```

Example:

```
#include<iostream.h>
#include<conio.h>
class employee
{
public:
int id;
int salary  // data members
void sal( )  // member function
{
  cout<<"enter salary: ";
  cin>>salary;
  cout<<"salary: "<<salary;
```

```
}
};
 void main( )
 {
  clrscr( );
  employee emp1; //creating an object of employee
  emp1.sal( );
  getch( );
 }
```

Output:

Enter salary: 4500

Salary: 4500

2. Explain Dynamic memory allocation

Every C++ program need some data as input, that input data is processed and given as output. This data is stored in variables and these variables are allocated some space in memory. Memory in C++ program is divided into two parts:

stack: All variables declared inside any function takes up memory from the stack.

heap: It is the unused memory of the program and can be used to dynamically allocate the memory at runtime.

There are two ways via which memories can be allocated for storing data. The two ways are:

Compile time allocation or static allocation of memory: where the memory for named variables is allocated by the compiler. Exact size and storage must be known at compile time and for array declaration, the size has to be constant. Stack memory is used by the compiler for static memory allocation.

Runtime allocation or dynamic allocation of memory: where the memory is allocated at runtime and the allocation of memory space is done dynamically within the program run and the memory segment is known as **a heap** or the free store. In this case, the exact space or number of the item does not have to be known by the compiler in advance. Pointers play a major role in this case.

Programmers can dynamically allocate storage space while the program is running, but programmers cannot create new variable names "on the fly", and for this reason, dynamic allocation requires two criteria:

- Creating the dynamic space in memory
- Storing its address in a pointer (so that space can be accessed)

Memory de-allocation is also a part of this concept where the "clean-up" of space is done for variables or other data storage. It is the job of the programmer to de-allocate dynamically created space. For de-allocating dynamic memory, we use the delete operator. In other words, dynamic memory Allocation refers to performing memory management for dynamic memory allocation manually.

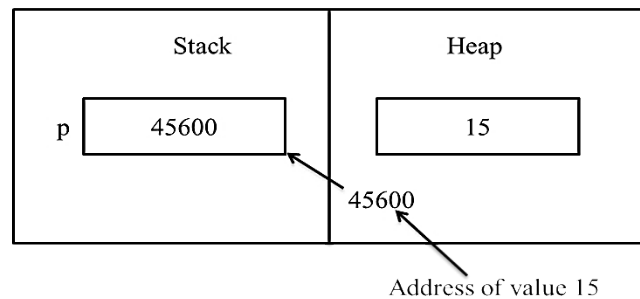
Syntax for Dynamic memory allocation using "new" operator

int * p; // declares a pointer p

p = new int;   // dynamically allocate an int and stores the address in p

Syntax for memory de allocation using "delete" operator

delete p; // de-allocate memory of p



Example:

```
#include <iostream.h>
void main( )
{
   int *p;
   p = new int(15);
   cout << "Value is: " << *p << endl;
   delete p;
}
```

In the above example, p is a pointer variable. When new operator is used an integer value 15 is stored in heap area and the address of that location will be assigned to p. when delete operator is used the space allotted for storing the address will be freed.

3. What is a constructor? Explain different types of constructors with examples

A constructor is a special member function that initializes an object of its class. A constructor has the same name as the class and no return value. In C++, constructor is automatically called when object is created.

Features of a constructor:

1. Constructor has same name of its class
2. Constructor does not have return type
3. Constructor is automatically called when object is created for its class
4. If no constructor is specified, C++ compiler generates a default compiler with no arguments and empty body.

In C++, there are three types of constructors. They are

1. Default constructor
2. Parameterized constructor
3. Copy constructor

Default constructor: If the constructor has no arguments, it is called default constructor.

Example:

```
#include <iostream.h>
class construct
{
public:
int a, b;
construct( ) // default constructor
{
a=10;
b=20;
}
};
void main ( )
{
construct c;
cout<<"a="<<c.a<<""b="<<c.b;
getch( );
}
```

Output: a=10 b=20

Parameterized Constructor: A default constructor does not have any parameters, but if needed, a constructor can have parameters. These constructors are called Parameterized Constructors.

Example:

```
#include <iostream.h>
class construct
{
public:
int a, b;
public:
construct(int x, int y) // Parameterized constructor
{
a=x;
b=y;
}
void display( )
{
cout<<"values are:"<<a<<b;
}
};
void main( )
{
construct c(10,20);
c.display( );
getch( );
}
```

Output: values are: 10 20

Copy Constructor: A copy constructor is a member function which initializes an object using another object of the same class. Copy constructor use objects as parameters.

Example:

```
#include <iostream.h>
class construct
{
public:
int x, y;
public:
construct(int x1, int y1) // Parameterized constructor
{
x=x1;
y=y1;
}
construct(construct &obj)  // copy constructor
{
a=obj.a;
b=obj.b;
}
void display( )
{
cout<<"a="<<a<<"b="<<b;
}
void main( )
{
construct c1(10,20); //invokes Parameterized constructor
construct c2(c1);  // invokes copy constructor
c1.show( );
c2.show( );
getch( );
}
```

Output: a=10 b=20

a=10 b=20

4. Access modifiers or specifiers in C++

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labels public, private, and protected sections within the class body. The keywords **public**, **private**, and **protected** are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

```
class Base {

  public:

    // public members go here

    protected:

   // protected members go here

  private:

  // private members go here

 };
```

Public Access Specifier:

A public member is accessible from anywhere outside the class but within a program (within the same class, or outside of the class).

Private Access Specifier:

Private class members are accessible with the class and it is not accessible outside the class. If someone tries to access outside the class, it gives compile time error. By default class variables and member functions are private.

Example for Public and Private Access Specifiers:

```
#include<iostream.h>
#include<conio.h>
class A
{
private:
int a;
public:
int b;
public:
```

```
void show( )
{
a=10 ;
b=20;  //Every members can be access here, same class
cout<<"\nAccessing variable within the class"<<endl;
cout<<"Value of a: "<<a<<endl;
cout<<"Value of b: "<<b<<endl;
}
};
void main( )
{
clrscr( );
A obj; // create object
obj.show( );
cout<<"\nAccessing variable outside the class"<<endl;
//'a' cannot be accessed as it is private
//cout<<"value of a: "<<obj.a<<endl;
//'b' is public as can be accessed from any where
cout<<"value of b: "<<obj.b<<endl;
getch( );
}
```

Output:
Accessing variable within the class
value of a: 10
value of b: 20
value of c: 30
Accessing variable outside the class
Value of b: 20

Protected Access Specifier:

A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

```
#include<iostream.h>
#include<conio.h>
Class parent
{
protected:
int id_protectd;
};
Class child: public parent
{
        Public:
        Void setId(int id)
```

```
{
Id_protected=id;
}
        Void displayId( )
        {
        Cout<<"Id_protected is:"<<id_protected<<endl;
        }
};
Void main( )
{
Child obj;
Obj.setId(12);
Obj.displayId( );
getch( );
}
```

Output: Id_protected is:12

5. Explain **Function overloading or Method overloading** with example

Whenever same function or method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as function or method overloading. Function overloading can be considered as an example of polymorphism feature in C++.

An overloaded function can have

1. Different types of parameters
2. Different number of parameters
3. Different order of parameters

The function get( ) can take the following forms when it is overloaded

1. get(int i)
2. get(double a)
3. get(int i, double a)
4. get(double a, double a)

Example:

```
#include <iostream.h>
class overload {
public:
   int get(int i)
```

```
{
      return i;
   }
   double get(double d)
{
      return d;
   }
};
void main( )
{
   overload obj;
   cout<<obj.get(10)<<endl;
   cout<<obj.get(25.32);
  getch( );
}
```

Output:
10
25.32

The main advantage of function overloading is to improve the code readability and allows code reusability. In the above program the function get ( ) is used to read an integer later a double value and the function is said to be overloaded.

6. Explain **Operator Overloading** in C++

Usually an operator is used to perform a particular operation on operands belong to basic data type. Operators cannot be used on user defined data types. In C++, using operator overloading, operators can be used on user defined data types also. In other words, operators can be used for some other work along with their natural functionality. All the operators can be overloaded except the following

1. **: :** Scope resolution operator
2. **.** and **.*** member selection and member selection through pointer to function operators
3. Sizeof ( )
4. **?:** Conditional operator

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
... .. ...
public
returnType   operator  symbol (arguments)
{
... .. ...
}
... .. ...
};
```

1. Here, returnType is the return type of the function.
2. The returnType of the function is followed by operator keyword.
3. Symbol is the operator symbol you want to overload. Like: +, <, -, ++
4. You can pass arguments to the operator function in similar way as functions.

Example: Binary operator (= =) overloading

```
#include<iostream.h>
class test
{
int a;
public:
void get( )
{
cin>>a;
}
void operator = = (test t2)
{
if(a= = t2.a) cout<<"both are equal"<<endl;
else cout<<"both are not equal";
}
};
void main( )
{
test  t1,t2;
cout<<"enter t1 object a value"<<endl;
t1.get( );
cout<<"enter t2 object a value"<<endl;
t2.get( );
t1= = t2;
getch( );
}
```

Output: Enter t1 object a value 9
Enter t2 object a value 9
Both are equal

Example: Unary operator (++,- -) overloading

```
#include<iostream.h>
class test
{
int a;
public:
test( )
{
a=0;
}
void operator + +( )
{
a++;
}
```

```
void operator - -( )
{
a- -;
}
void show( )
{
cout<<"a="<<a<<endl;
}
};
void main( )
{
test t;
t++;
t.show( );
t- -;
t.show( );
getch( );
}
```

Output:
a=1
a=0

Explain inheritance with example

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class: The class whose properties are inherited by sub class is called Base Class or Super class

Modes of Inheritance

Public mode: If we derive a sub class from a public base class, then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.

Protected mode: If we derive a sub class from a protected base class, then both public member and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.

Private mode: If we derive a sub class from a Private base class, then both public member and protected members of the base class will become Private in derived class. Private members of the base class will never get inherited in sub class.

The basic syntax of inheritance is:

class DerivedClass : accessSpecifier BaseClass

Access specifier can be public, protected and private. The default access specifier is private. Access specifiers affect accessibility of data members of base class from the derived class. In addition, it determines the accessibility of data members of base class outside the derived class.

There are different types of inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid (Virtual) Inheritance

Single inheritance:

Single inheritance represents a form of inheritance when there is only one base class and one derived class.

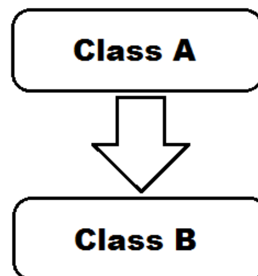


Fig: Single inheritance

Example for single inheritance

```
#include <iostream.h>

class Parent  //Base class

{
   public:
     int id_p;
};

class Child : public Parent   // Sub class inheriting from Base Class(Parent)

{
   public:
     int id_c;
};

void main( )   //main function

   {
```

```
        Child obj1;

        obj1.id_c = 18;
        obj1.id_p = 27;
        cout << "Child id is " <<  obj1.id_c << endl;
        cout << "Parent id is " <<  obj1.id_p << endl;

        getch( );
  }
```

Output:

Child id is 18
Parent id is 27

Multiple Inheritance

When a class is derived from two or more base classes, such inheritance is called Multiple Inheritance. It allows us to combine the features of several existing classes into a single class.

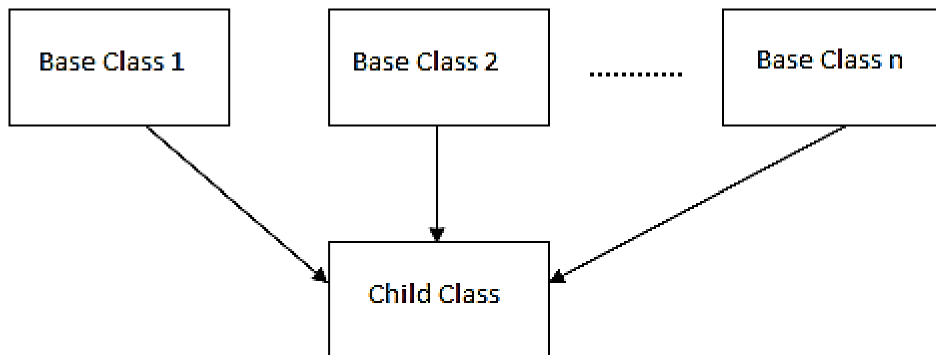


Fig: Multiple Inheritance

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

class A
{
  public:
    void display( )
    {
        cout <<"This is method of A";
    }
};
```

```
class B
{
  public:
    void display( )
    {
        cout <<"This is method of B";
    }
};

class C: public A, public B  // Multiple inheritance

{
  public:
B::display( );  //directing compiler to execute class B method
};

int main( )
{
    C sample;
    sample.display( );
    getch( );
    }
```

Output:

This is method of B

Multilevel Inheritance:

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

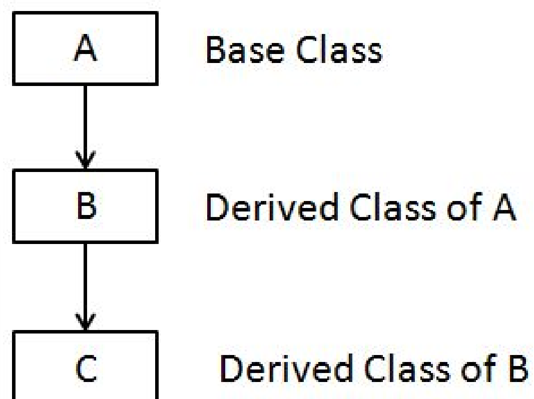


Fig: Multi level inheritance

Example for Multilevel inheritance

```
#include <iostream>

class A
{
   public:
     void display( )
     {
        cout<<"Base class content.";
     }
};

class B : public A
{

};

class C : public B // Multilevel inheritance
{

};

void main( )
{
   C obj;
   obj.display( );
   getch( );
}
```

Output:
Base class content

In this program, class C is derived from class B and B is derived from base class A. This is called multilevel inheritance.

Hierarchical Inheritance

When more than one classes are derived from a single base class, such inheritance is known as Hierarchical Inheritance.

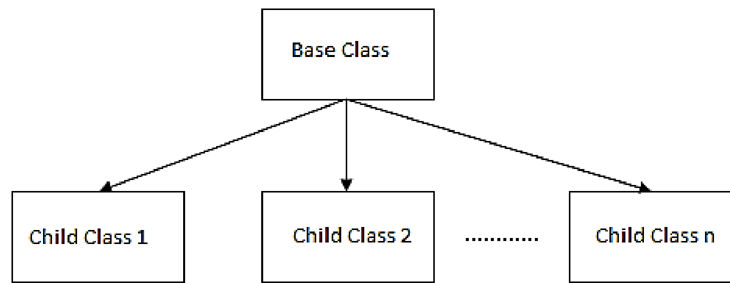


Fig: Hierarchical Inheritance

Hybrid (Virtual) Inheritance

Hybrid Inheritance is a method where one or more types of inheritance are combined together and used.

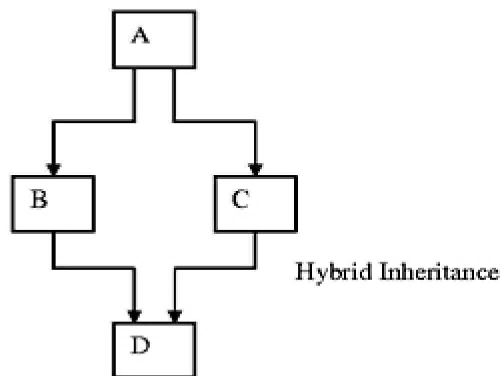


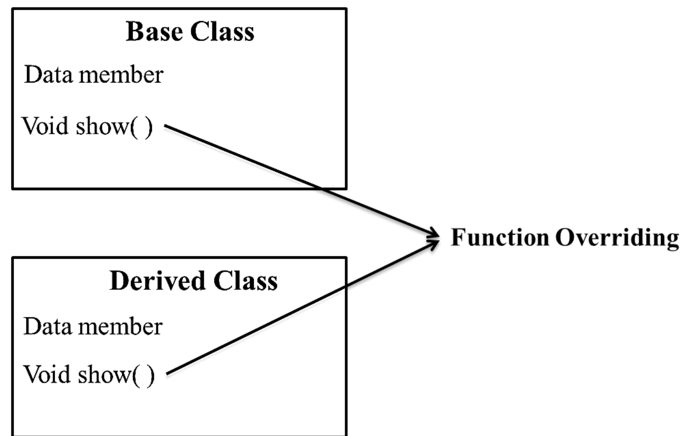
Fig: Hybrid or Virtual inheritance

Explain Function overriding or Method overriding with example

In C++, if a derived class defines same function as defined in its base class, it is known as function overriding. It is used to achieve runtime polymorphism. It enables to provide specific implementation of the function which is already provided by its base class. To override a function we must have the same function prototype in child class i.e. same data type and sequence of parameters.

Conditions for Function Overriding

1. Functions of both parent and child class must have the same name.
2. Functions must have the same argument list and return type.
3. A function declared static cannot be overridden.
4. If a function cannot be inherited, it cannot be overridden.



Example for Function or Method Overriding

```
#include<iostream.h>
class Base
{
 public:
 void show( )
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show( )  // Over ridden function
 {
  cout << "Derived Class";
 }
}

int main( )
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show( );     //Early Binding Occurs
}
Output: Base class
```

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding. On seeing Base class's pointer, compiler set call to Base class's show ( ) function, without knowing the actual object type.

Explain Virtual functions with example

Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform Late Binding on this function. Virtual Keyword is used to make a member function of the base class Virtual. They are mainly used to achieve Runtime polymorphism.

Rules for Virtual Functions

1. They must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.

Example for Virtual Function

```
#include<iostream.h>
class Base
{
 public:
virtual void show( )  // virtual key word is used
 {
  cout << "Base class";
 }
};
class Derived: public Base
{
 public:
 void show( )  // Over ridden function
 {
  cout << "Derived Class";
 }
}

int main( )
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show( );     //late or dynamic Binding Occurs
}
```

Output: Derived class

In the above program, base class pointer 'b' contains the address of derived class object 'd'. When the function is called with base class pointer 'b', compiler checks the content of the pointer instead of the type. Then late binding (Runtime) is done. It executes the derived class function and prints 'Derived class'.

Explain Pure Virtual Function with example

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. Pure virtual function doesn't have body or implementation. Definition of pure virtual function is written in derived class. Pure virtual function is also known as abstract function or "do-nothing" function. A class with at least one pure virtual function or abstract function is called abstract class. Objects can't be created for abstract class. Member functions of abstract class will be invoked by derived class object.

Syntax for Pure Virtual Functions:

virtual return_type function_name(arguments)=0;

Example for Pure Virtual Function

```
#include< iostream.h>
class base
{
public:
virtual void show( )=0; // Pure Virtual Function
};
Class derived: public base
{
public:
void show( )
{
cout << "Derived Class";
}
};
void main( )
{
Derived d;
d.show( );
getch( );
}
```

Output: Derived Class

Explain about stream classes

The stream is a flow of data, measured in bytes, in sequence and acts as an inter-mediator between I/O devices and the user. The C++ supports a number of I/O operations to perform read and write operations. These C++ I/O functions help the user to work with different types of devices such as keyboard, disk, tape drivers, etc. If data is received from input devices in sequence, then it is called as source stream, and when the data is passed to output devices, then it is called as destination stream.

The input stream receives data from keyboard or storage devices such as hard disk, floppy disk, etc. The data present in output stream is passed on to the output devices such as monitor or printer according to the user's choice. C++ has number of classes that work with console and file operations. These classes are known as stream classes. All these classes are declared in the header file iostream.h. The file iostream.h must be included in the program, if we are using the functions of these classes.

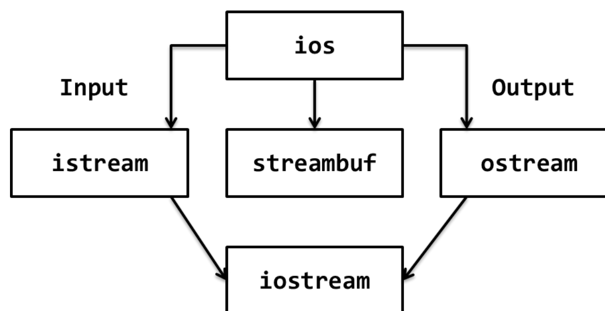


Fig: Stream classes

The classes istream and ostream are derived classes of base class ios. The ios class contains another derived class streambuf. The streambuf places the buffer. The member function of streambuf class handles the buffer by providing the facilities to flush clear and pour the buffer. The class iostream is derived from the classes istream and ostream by using multiple inheritance. The ios class is a virtual class and it is present to avoid ambiguity that frequently appears in multiple inheritance. Two kinds of streams are generally discussed in C++. They are

Input Stream

The input stream reads operation through keyboard. It uses cin as object. The cin statement uses >> (extraction operator) before a variable name. The cin statement is used to read data through the input device. Its syntax and example are as follows.

Syntax:

cin>>variable;

Example:

int x,y,z;

cin>> x>> y>> z;

Output Streams

The output streams manage output of the stream, that is, display contents of variables on the screen.

It uses << insertion operator before the variable name. It uses the cout object to perform console write operation. The syntax and example of cout statement are as follows.

Syntax:

cout<<variable

Example:

cout<<x <<y <<z;

The above statement displays the contents of these variables on the screen.

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are ued by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streanbuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

Fig: Stream classes

Pre-Defined Streams in C++

C++ has a number of pre-defined streams. These pre-defined streams are also called as standard I/O objects. These streams are automatically activated when program execution starts.

Object	Functionality
cin	Standard input, usually keyboard, corresponding to stdin in C. It handles input from input devices usually from keyboard.
cout	Standard output, usually screen, corresponding to stdout in C. It passes data to output devices such as monitors and printers. Thus, it controls output.
clog	A fully buffered version of cerr (no C equivalent). It controls error messages that are passed from buffer to the standard error device
cerr	Standard error output, usually screen, corresponding to stderr in C. It controls the unbuffered output data. It catches the errors and passes to standard error device monitor

Fig: Pre-Defined Streams in C++

Explain Exception handling in C++ with example

Exception handling or Error Handling is the process of handling errors and exceptions in such a way that they do not disturb normal operation of the program.

In C++, errors are classified into two types. They are

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors: Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import etc.

Run Time Errors: They are also known as exceptions. An exception caught during run time creates serious problems.

Errors disturb normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not disturb normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this exception handling technics are used.

In C++, Error handling is done by three keywords:-

1. Try
2. Catch
3. Throw

1. Try: Try block is intended to throw exceptions, which is followed by catch blocks. Only one try block.

2. Catch: Catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks.

3. Throw: It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.

Syntax:

```
Try
{
//code
throw parameter;
}
catch(exceptionname ex)
{
//code to handle exception
}
```

Example for Exception handling

```
#include <iostream>
#include<conio.h>
void main( )
{
int a,b,c;
try   //try block activates exception handling
{
cout<<"Enter two numbers for division";
cin>>a>>b;
if(b= =0) throw 1;
cout<<" Division is:"<<a/b;
}
catch(int i) //catches exception
{
cout<<" Division by zero not possible";
}
getch( );
}
```

Output:
Enter two numbers for division 15 5
Division is:3

Enter two numbers for division 15 0
Division by zero not possible

Multiple Catch Statements

A single try statement can have multiple catch statements. Execution of particular catch block depends on the type of exception thrown by the throw keyword. If throw keyword send exception of integer type, catch block with integer parameter will get execute.

```
#include<iostream>
void main( )
{
int a, b;
cout<<"Enter a and b values: ";
cin>>a>>b;
try
{
if(b = = 0)
throw b;
else if(b < 0)
throw "b cannot be negative";
else
cout<<"Result of a/b = "<<(a/b);
}
catch(int b)
{
cout<<"b cannot be zero";
}
catch(const char *msg)
{
cout<<msg;
}
getch( );
}
```

Output:
Enter a and b values: 10 0
b cannot be zero
Enter a and b values: 10 –5
b cannot be negative