Programming in C/C++

UNIT-I

C language: Introduction ; Tokens; Keywords; Identifier ; Variables; Constants; Operators ; Expression; Data types; Operator precedence

Statement: Input statement, Output statement, Conditional and Unconditional Control Statement – Looping Statement: while, do-while, for – nested loop – Arrays.

Overview of C++: Object oriented programming, Introducing C++ classes, Concepts of object oriented programming. Classes & Objects : Classes, Structure & classes, Union & Classes, Friend function, Friend classes, Inline function, Scope resolution operator, Static class members: Static data member, Static member function, Passing objects to function, Returning objects, Object assignment.

UNIT-II

Array and Pointers references: Array of objects, Pointers to object, Type checking C++ pointers, The This pointer, Pointer to derived types, Pointer to class members, References: Reference parameter, Passing references to objects, Returning reference, Independent reference, C++ ’s dynamic allocation operators, Initializing allocated memory, Allocating Array, Allocating objects.

Constructor & Destructor: Introduction, Constructor, Parameterized constructor, Multiple constructor in a class, Constructor with default argument, Copy constructor, Default Argument, Constructing two dimensional Array, Destructor.

UNIT-III

Function & operator overloading : Function overloading, Overloading constructor function finding the address of an overloaded function, Operator Overloading: Creating a member operator function, Creating Prefix & Postfix forms of the increment & decrement operation, Overloading the shorthand operation (i.e. +=,-= etc), Operator overloading restrictions, Operator overloading using friend function, Overloading New & Delete, Overloading some special operators, Overloading [ ], ( ), -, comma operator, Overloading << .

UNIT-IV

Inheritance : Base class Access control, Inheritance & protected members, Protected base class inheritance, Inheriting multiple base classes, Constructors, destructors & Inheritance, When constructor & destructor function are executed, Passing parameters to base class constructors, Granting access. Virtual functions & Polymorphism: Virtual base classes; Virtual function, Pure Virtual functions, early Vs. late binding

UNIT-V

String Handling: String declaration; String library functions; String Manipulation; Creating string objects, manipulating string objects, relational operators, string characteristics, Comparing and swapping
Sorting: Bubble sort, Selection sort, Insertion sort
Searching: Linear search, Binary search

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

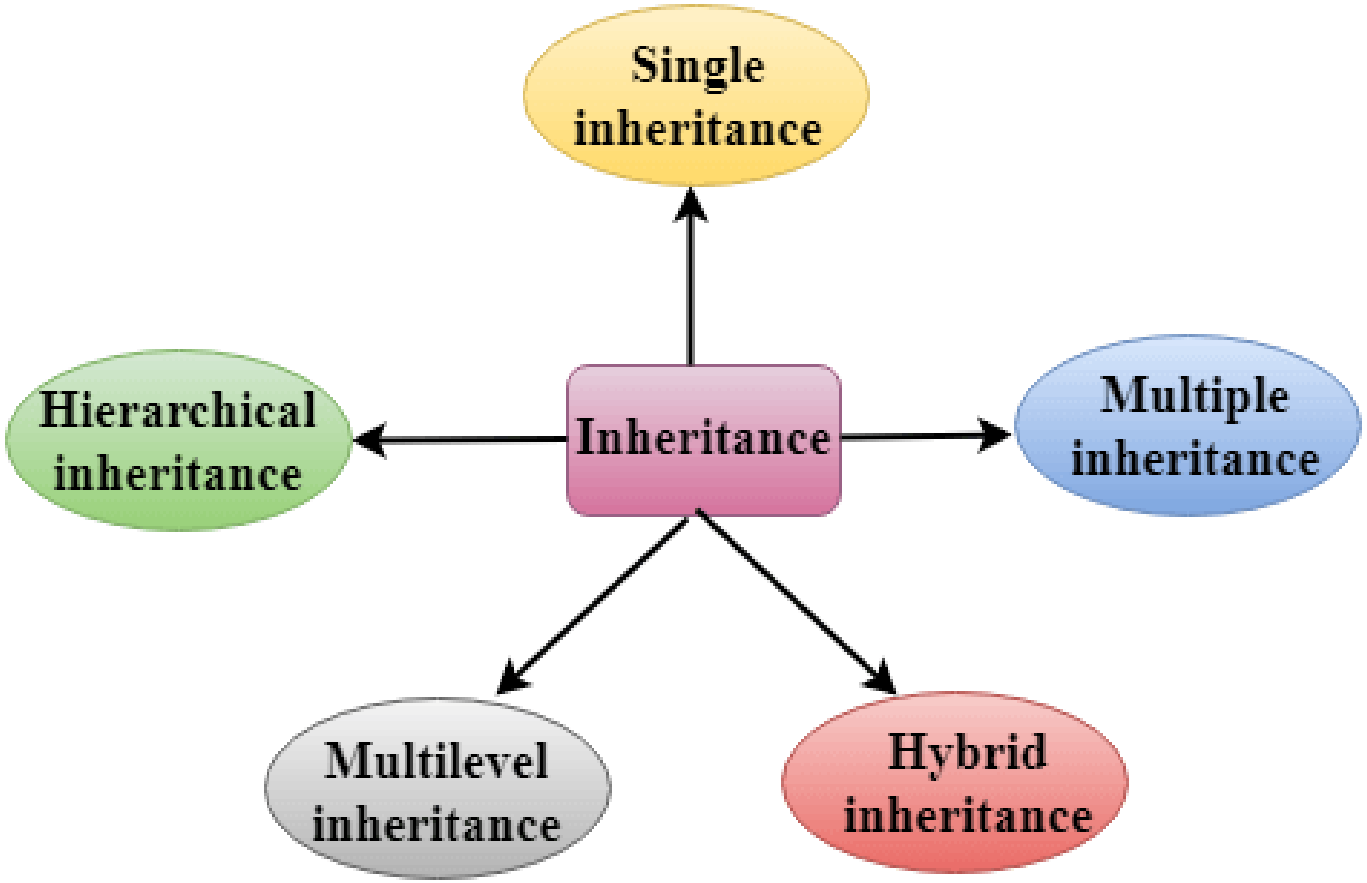
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
1.class derived_class_name :: visibility-mode base_class_name
2.{
3.    // body of the derived class.
4.}
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

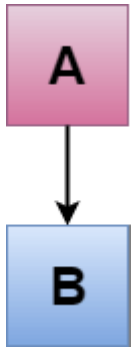
Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.

Where 'A' is the base class, and 'B' is the derived class.



C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
1.#include <iostream.h>
#include <conio.h>
 class Account
 {
public:
 float salary = 60000;

 };
 class Programmer: public Account
{
  public:
 float bonus = 5000;
  };
void main(void)

 {
Programmer p1;
 cout<<"Salary: "<<p1.salary<<endl;
cout<<"Bonus: "<<p1.bonus<<endl;

}
```

Output:
Salary: 60000
Bonus: 5000

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

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

class Animal {
  public:
void eat() {
   cout<<"Eating..."<<endl;
 }
 };
 class Dog: public Animal
  {
      public:
    void bark(){
  cout<<"Barking...";
   }
  };
int main(void) {
 Dog d1;
d1.eat();
d1.bark();
 return 0;
}
```

Output
Eating...
Barking..

Let's see a simple example.

Output:

Multiplication of a and b is : 20

```
#include <iostream>
class A
{
  int a = 4;
 int b = 5;
 public:
  int mul()
  {
      int c = a*b;
       return c;
  }
};

class B : private A
{
   public:
   void display()
   {
       int result = mul();
       std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
   }
};
int main()
{
   B b;
 b.display();

   return 0;
}
```

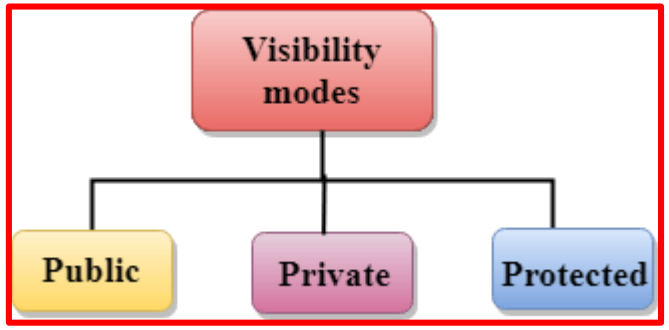
How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**.

The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:



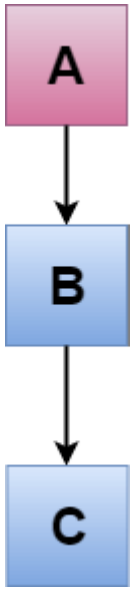
- **Public**: When the member is declared as public, it is accessible to all the functions of the program.

- **Private**: When the member is declared as private, it is accessible within the class only.

- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++.

Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

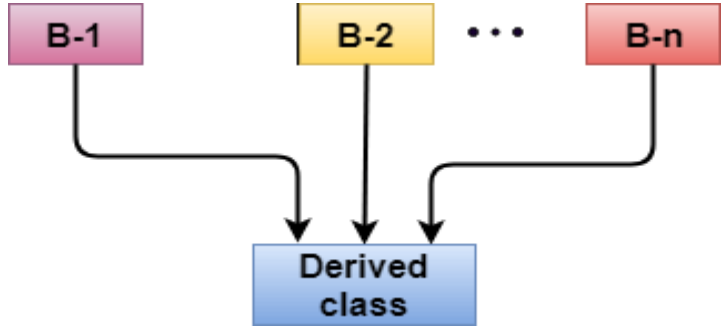
```
#include <iostream.h>
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
   };
   class BabyDog: public Dog
   {      public:
     void weep() {
    cout<<"Weeping...";
     }
   }; int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

Eating...
Barking...
Weeping..

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class

```
class D : visibility B-1, visibility B-2, ?
{

   // Body of the class;

 }
```

Let's see a simple example of multiple inheritance.

```
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

```
#include <iostream.h>
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};
class B
{
   protected:
   int b;
   public:
   void get_b(int n)
   {
      b = n;
   }
};
class C : public A,public B
{
  public:
   void display()
   {
      std::cout << "The value of a is : " <<a<< std::endl;
      std::cout << "The value of b is : " <<b<< std::endl;
      cout<<"Addition of a and b is : "<<a+b;
   }
};
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.
Let's understand this through an example:

```
#include <iostream.h>
class A
{     public:
   void display()
   {
      std::cout << "Class A" <<
std::endl;
   }
};
class B
{
   public:
   void display()
   {        std::cout << "Class B" <<
std::endl;
   }
};
class C : public A, public B
{
   void view()
   {
      display();
   }
};
```

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

Output:

error: reference to 'display' is ambiguous
display();

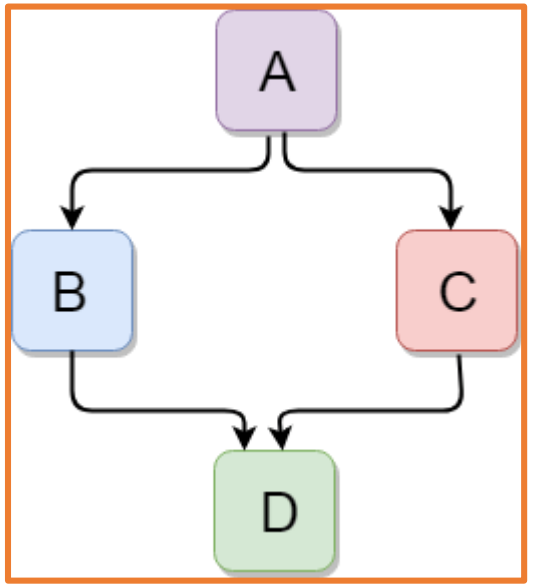
•The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   void view()
   {
      A :: display();        // Calling the display() function of class A.
      B :: display();        // Calling the display() function of class B.

   }
};
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
      std::cout << "Enter the value of 'a' :
" << std::endl;
      cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
       std::cout << "Enter the value of 'b' :
" << std::endl;
      cin>>b;
   }
};

class C
{
   protected:
   int c;
   public:
   void get_c()
   {
      std::cout << "Enter the value of c is :
" << std::endl;
      cin>>c;
   }
};

class D : public B, public C
{
   protected:
   int d;
   public:
   void mul()
   {
       get_a();
       get_b();
       get_c();
       std::cout << "Multiplication of a,b,c
is : " <<a*b*c<< std::endl;
   }
};

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

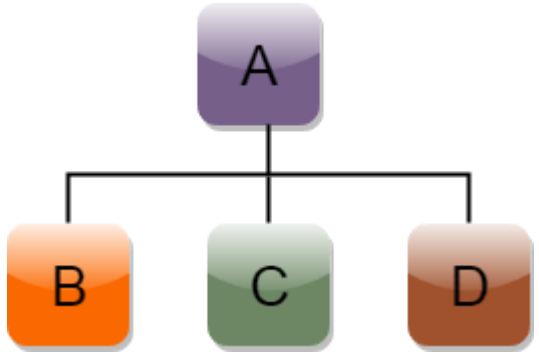
Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

Syntax of Hierarchical inheritance:




```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream.h>
 class Employee
 {
   public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

Output:

Default Constructor Invoked
Default Constructor Invoked

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream.h>
 class Employee
 {
   public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
      float salary;
      Employee(int i, string n, float s)
       {
          id = i;
          name = n;
          salary = s;
       }
      void display()
       {
          cout<<id<<"  "<<name<<"  "<<salary<<endl;
       }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of
Employee
   Employee e2=Employee(102, "Nakul", 59000);
   e1.display();
   e2.display();
   return 0;
}
```

Output:

101 Sonoo 890000
102 Nakul 59000

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
       Employee()
       {
          cout<<"Constructor Invoked"<<endl;
       }
       ~Employee()
       {
          cout<<"Destructor Invoked"<<endl;
       }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

.

```
#include <iostream.h>

class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
      {
          this->id = id;
         this->name = name;
         this->salary = salary;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of
Employee
   Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
   e1.display();
   e2.display();
   return 0;
}
```

Output:

101 Sonoo 890000
102 Nakul 59000

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
   friend data_type function_name(argument/s);          //
syntax of friend function.
};
```

Characteristics of a Friend function:

•The function is not in the scope of the class to which it has been declared as a friend.

•It cannot be called using the object as it is not in the scope of that class.

•It can be invoked like a normal function without using the object.

•It cannot access the member names directly and has to use an object name and dot membership operator with the member name.

•It can be declared either in the private or the public part.

Programming in C/C++

Programming in C/C++

Programming in C/C++

Programming in C/C++