C++ OOPs Concepts

C++ Object Class

C++ OOPs Concepts

C++ Object Class

C++ Constructor

C++ Copy Constructor

C++ Destructor

C++ this Pointer

C++ static

C++ Structs

C++ Enumeration

C++ Friend Function

C++ Math Functions

C++ Inheritance

C++ Inheritance

C++ Aggregation

C++ Polymorphism

C++ Polymorphism

C++ Overloading

C++ Overriding

C++ Virtual Function

C++ Abstraction

C++ Interfaces

C++ Data Abstraction

C++ OOPs Concepts

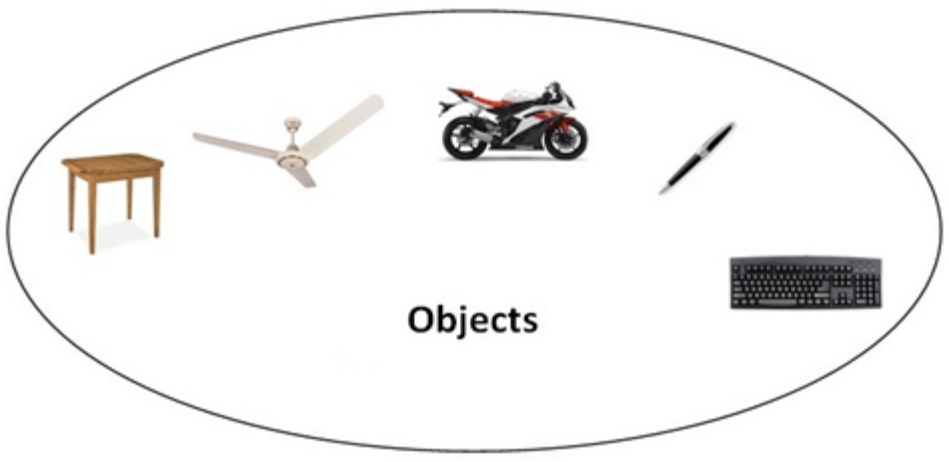
The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:



- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

```
1. Student s1;  //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
1. class Student
2. {
3.     public:
4.     int id;  //field or data member
5.     float salary; //field or data member
6.     String name;//field or data member
7. }
```

C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
1. #include <iostream>
```

```
2.  using namespace std;
3.  class Student {
4.    public:
5.       int id;//data member (also instance variable)
6.       string name;//data member(also instance variable)
7.  };
8.  int main() {
9.      Student s1; //creating an object of Student
10.     s1.id = 201;
11.     s1.name = "Sonoo Jaiswal";
12.     cout<<s1.id<<endl;
13.     cout<<s1.name<<endl;
14.     return 0;
15. }
```

Output:

```
201
Sonoo Jaiswal
```

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Student {
4.    public:
5.       int id;//data member (also instance variable)
6.       string name;//data member(also instance variable)
7.       void insert(int i, string n)
8.       {
9.           id = i;
10.          name = n;
11.      }
12.      void display()
13.      {
14.          cout<<id<<"  "<<name<<endl;
15.      }
16. };
17. int main(void) {
18.     Student s1; //creating an object of Student
19.     Student s2; //creating an object of Student
20.     s1.insert(201, "Sonoo");
21.     s2.insert(202, "Nakul");
22.     s1.display();
23.     s2.display();
24.     return 0;
25. }
```

Output:

```
201  Sonoo
202  Nakul
```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
1.  #include <iostream>
```

```
2.  using namespace std;
3.  class Employee {
4.    public:
5.       int id;//data member (also instance variable)
6.       string name;//data member(also instance variable)
7.       float salary;
8.       void insert(int i, string n, float s)
9.        {
10.           id = i;
11.           name = n;
12.           salary = s;
13.       }
14.      void display()
15.       {
16.           cout<<id<<"  "<<name<<"  "<<salary<<endl;
17.       }
18. };
19. int main(void) {
20.    Employee e1; //creating an object of Employee
21.    Employee e2; //creating an object of Employee
22.    e1.insert(201, "Sonoo",990000);
23.    e2.insert(202, "Nakul", 29000);
24.    e1.display();
25.    e2.display();
26.    return 0;
27. }
```

Output:

```
201  Sonoo  990000
202  Nakul  29000
```

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.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Employee
4.   {
5.     public:
6.        Employee()
7.        {
8.            cout<<"Default Constructor Invoked"<<endl;
9.        }
10. };
```

```
11. int main(void)
12. {
13.     Employee e1; //creating an object of Employee
14.     Employee e2;
15.     return 0;
16. }
```

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>
using namespace std;
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 (~).

> ***Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.***

C++ Constructor and Destructor Example

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

```
1.  #include <iostream>
2.  using namespace std;
3.  class Employee
4.  {
5.    public:
6.        Employee()
7.        {
8.            cout<<"Constructor Invoked"<<endl;
9.        }
10.       ~Employee()
```

```
11.         {
12.             cout<<"Destructor Invoked"<<endl;
13.         }
14. };
15. int main(void)
16. {
17.     Employee e1; //creating an object of Employee
18.     Employee e2; //creating an object of Employee
19.     return 0;
20. }
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

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

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Employee {
4.    public:
5.        int id; //data member (also instance variable)
6.        string name; //data member(also instance variable)
7.        float salary;
8.        Employee(int id, string name, float salary)
9.         {
10.            this->id = id;
11.           this->name = name;
12.           this->salary = salary;
13.       }
14.      void display()
15.       {
16.           cout<<id<<"  "<<name<<"  "<<salary<<endl;
17.       }
18. };
19. int main(void) {
20.     Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.     Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22.     e1.display();
23.     e2.display();
24.     return 0;
25. }
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

C++ static field example

Let's see the simple example of static field in C++.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Account {
4.    public:
5.       int accno; //data member (also instance variable)
6.       string name; //data member(also instance variable)
7.       static float rateOfInterest;
8.       Account(int accno, string name)
9.        {
10.           this->accno = accno;
11.          this->name = name;
12.       }
13.       void display()
14.       {
15.           cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
16.       }
17. };
18. float Account::rateOfInterest=6.5;
19. int main(void) {
20.    Account a1 =Account(201, "Sanjay"); //creating an object of Employee
21.    Account a2=Account(202, "Nakul"); //creating an object of Employee
22.    a1.display();
23.    a2.display();
24.    return 0;
25. }
```

Output:

```
201 Sanjay 6.5
202 Nakul 6.5
```

C++ static field example: Counting Objects

Let's see another example of static keyword in C++ which counts the objects.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Account {
4.    public:
5.       int accno; //data member (also instance variable)
6.       string name;
7.       static int count;
8.       Account(int accno, string name)
9.        {
10.           this->accno = accno;
11.          this->name = name;
12.          count++;
13.       }
14.      void display()
15.       {
16.          cout<<accno<<" "<<name<<endl;
17.       }
18. };
19. int Account::count=0;
20. int main(void) {
21.    Account a1 =Account(201, "Sanjay"); //creating an object of Account
22.    Account a2=Account(202, "Nakul");
23.     Account a3=Account(203, "Ranjana");
24.    a1.display();
25.    a2.display();
26.    a3.display();
27.    cout<<"Total Objects are: "<<Account::count;
28.    return 0;
29. }
```

Output:

```
201 Sanjay
202 Nakul
203 Ranjana
Total Objects are: 3
```

C++ Structs

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.

Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

The Syntax Of Structure

```
1.  struct structure_name
2.  {
3.       // member declarations.
4.  }
```

In the above declaration, a structure is declared by preceding the **struct keyword** followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. **Consider the following situation:**

```
1. struct Student
2. {
3.     char name[20];
4.      int id;
5.      int age;
6. }
```

In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

Student s;

Here, s is a structure variable of type **Student**. When the structure variable is created, the memory will be allocated. Student structure contains one char variable and two integer variable. Therefore, the memory for one char variable is 1 byte and two ints will be 2*4 = 8. The total memory occupied by the s variable is 9 byte.

How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example:

```
1. s.id = 4;
```

In the above statement, we are accessing the id field of the structure Student by using the **dot(.)** operator and assigns the value 4 to the id field.

C++ Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

```
1. #include <iostream>
2. using namespace std;
3.  struct Rectangle
4. {
5.    int width, height;
6.
7.  };
8. int main(void) {
9.     struct Rectangle rec;
10.    rec.width=8;
11.    rec.height=5;
12.   cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
13. return 0;
14. }
```

Output:

```
Area of Rectangle is: 40
```

C++ Struct Example: Using Constructor and Method

Let's see another example of struct where we are using the constructor to initialize data and method to calculate the area of rectangle.

```
1. #include <iostream>
2. using namespace std;
3.  struct Rectangle    {
4.    int width, height;
```

```
5.    Rectangle(int w, int h)
6.     {
7.         width = w;
8.         height = h;
9.     }
10.  void areaOfRectangle() {
11.    cout<<"Area of Rectangle is: "<<(width*height); }
12. };
13. int main(void) {
14.    struct Rectangle rec=Rectangle(4,6);
15.    rec.areaOfRectangle();
16.   return 0;
17. }
```

Output:

```
Area of Rectangle is: 24
```

Structure v/s Class

Structure	Class
If access specifier is not declared explicitly, then by default access specifier will be public.	If access specifier is not declared explicitly, then by default access specifier will be private.

Syntax of Structure:
 	
struct structure_name
{
// body of the structure.
}
	Syntax of Class:
	
	class class_name
	{
	// body of the class.
	}

The instance of the structure is known as "Structure variable".	The instance of the class is known as "Object of the class".

C++ Enumeration

Enum in C++ is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The C++ enum constants are static and final implicitly.

C++ Enums can be thought of as classes that have fixed set of constants.

Points to remember for C++ Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

---

C++ Enumeration Example

Let's see the simple example of enum data type used in C++ program.

```
1. #include <iostream>
2. using namespace std;
```

```
3.  enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
4.  int main()
5.  {
6.      week day;
7.      day = Friday;
8.      cout << "Day: " << day+1<<endl;
9.      return 0;
10. }
```

Output:

```
Day: 5
```

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++

```
1.  class class_name
2.  {
3.      friend data_type function_name(argument/s);          // syntax of friend function.
4.  };
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

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.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Box
4.  {
5.     private:
6.        int length;
7.     public:
8.        Box(): length(0) { }
9.        friend int printLength(Box); //friend function
10. };
11. int printLength(Box b)
```

```
12. {
13.    b.length += 10;
14.     return b.length;
15. }
16. int main()
17. {
18.     Box b;
19.     cout<<"Length of box: "<< printLength(b)<<endl;
20.     return 0;
21. }
```

Output:

```
Length of box: 10
```

Let's see a simple example when the function is friendly to two classes.

```
1.  #include <iostream>
2.  using namespace std;
3.  class B;          // forward declarartion.
4.  class A
5.  {
6.      int x;
7.      public:
8.      void setdata(int i)
9.      {
10.         x=i;
11.     }
12.     friend void min(A,B);         // friend function.
13. };
14. class B
15. {
16.     int y;
17.     public:
18.     void setdata(int i)
19.     {
20.         y=i;
21.     }
22.     friend void min(A,B);                  // friend function
23. };
24. void min(A a,B b)
25. {
26.     if(a.x<=b.y)
27.     std::cout << a.x << std::endl;
28.     else
29.     std::cout << b.y << std::endl;
30. }
31.    int main()
32. {
33.    A a;
34.    B b;
35.    a.setdata(10);
36.    b.setdata(20);
37.    min(a,b);
38.     return 0;
39.  }
```

Output:

```
10
```

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

Let's see a simple example of a friend class.

```
#include <iostream>

using namespace std;

class A
{
    int x =5;
    friend class B;          // friend class.
};
class B
{
  public:
    void display(A &a)
    {
        cout<<"value of x is : "<<a.x;
    }
};
int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

Output:

```
value of x is : 5
```

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

C++ Inheritance

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.

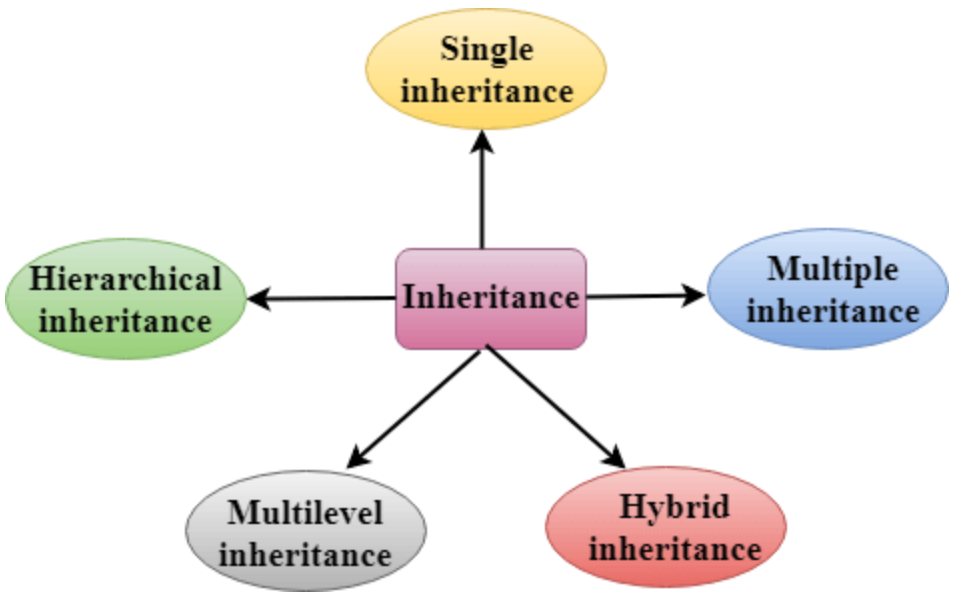
Advantage of C++ Inheritance

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:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes

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

The Syntax of Derived class:

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

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.

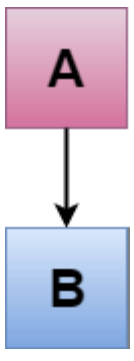
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions 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.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
  };
   class Programmer: public Account {
   public:
   float bonus = 5000;
  };
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

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>
using namespace std;
 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.

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

Output:

```
Multiplication of a and b is : 20
```

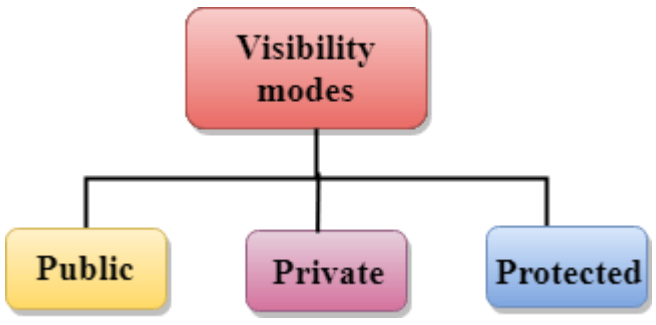
In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

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:



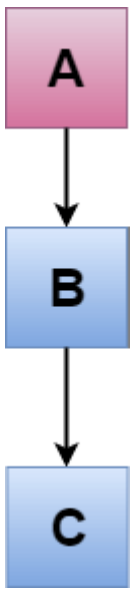
- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

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>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
  };
   class Dog: public Animal
   {
      public:
    void bark(){
   cout<<"Barking..."<<endl;
```

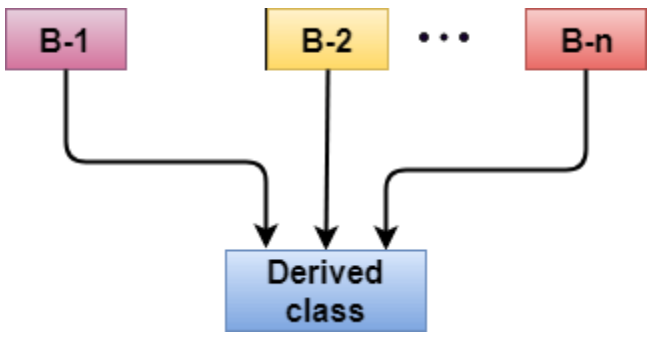
```
    }
};
class BabyDog: public Dog
{
    public:
  void weep() {
 cout<<"Weeping...";
  }
};
int main(void) {
  BabyDog d1;
  d1.eat();
  d1.bark();
   d1.weep();
   return 0;
}
```

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.

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

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>
using namespace std;
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()
```

```
22.    {
23.        display();
24.    }
25. };
26. int main()
27. {
28.    C c;
29.    c.display();
30.    return 0;
31. }
```

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:

```
1. class C : public A, public B
2. {
3.     void view()
4.     {
5.         A :: display();        // Calling the display() function of class A.
6.         B :: display();        // Calling the display() function of class B.
7.
8.     }
9. };
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

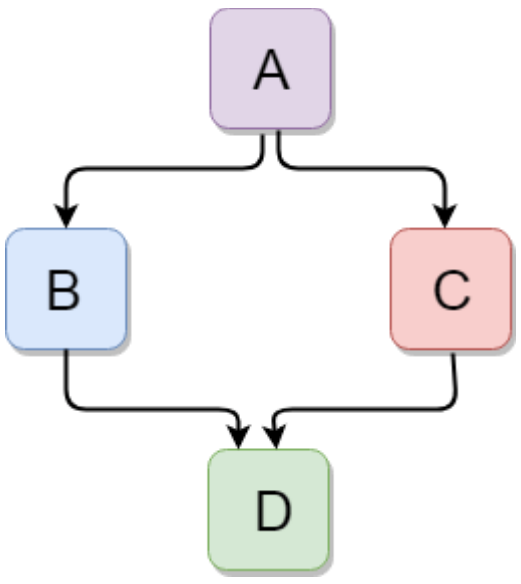
```
1. class A
2. {
3.    public:
4. void display()
5. {
6.    cout<<?Class A?;
7. }
8. } ;
9. class B
10. {
11.  public:
12.  void display()
13. {
14.  cout<<?Class B?;
15. }
16. } ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
1.  int main()
2.  {
3.      B b;
4.     b.display();              // Calling the display() function of B class.
5.     b.B :: display();       // Calling the display() function defined in B class.
6.  }
```

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:
```

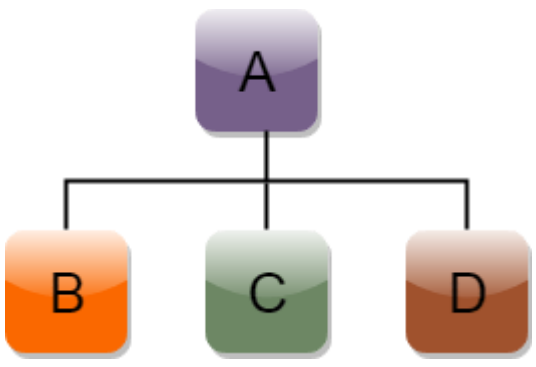
```
43.    void mul()
44.    {
45.        get_a();
46.        get_b();
47.        get_c();
48.        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
49.    }
50. };
51. int main()
52. {
53.    D d;
54.    d.mul();
55.    return 0;
56. }
```

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:

```
1. class A
2. {
3.     // body of the class A.
4. }
5. class B : public A
6. {
7.     // body of class B.
8. }
9. class C : public A
10. {
11.    // body of class C.
12. }
13. class D : public A
14. {
15.    // body of class D.
16. }
```

Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class Shape              // Declaration of base class.
4. {
5.     public:
```

```
6.      int a;
7.      int b;
8.      void get_data(int n,int m)
9.      {
10.         a= n;
11.         b = m;
12.     }
13. };
14. class Rectangle : public Shape  // inheriting Shape class
15. {
16.     public:
17.     int rect_area()
18.     {
19.         int result = a*b;
20.         return result;
21.     }
22. };
23. class Triangle : public Shape    // inheriting Shape class
24. {
25.     public:
26.     int triangle_area()
27.     {
28.         float result = 0.5*a*b;
29.         return result;
30.     }
31. };
32. int main()
33. {
34.     Rectangle r;
35.     Triangle t;
36.     int length,breadth,base,height;
37.     std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38.     cin>>length>>breadth;
39.     r.get_data(length,breadth);
40.     int m = r.rect_area();
41.     std::cout << "Area of the rectangle is : " <<m<< std::endl;
42.     std::cout << "Enter the base and height of the triangle: " << std::endl;
43.     cin>>base>>height;
44.     t.get_data(base,height);
45.     float n = t.triangle_area();
46.     std::cout <<"Area of the triangle is : "  << n<<std::endl;
47.     return 0;
48. }
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>
using namespace std;
class Address {
    public:
   string addressLine, city, state;
     Address(string addressLine, string city, string state)
    {
        this->addressLine = addressLine;
        this->city = city;
        this->state = state;
    }
};
class Employee
  {
      private:
      Address* address;  //Employee HAS-A Address
      public:
      int id;
      string name;
      Employee(int id, string name, Address* address)
     {
        this->id = id;
        this->name = name;
        this->address = address;
     }
   void display()
     {
        cout<<id <<" "<<name<< " "<<
          address->addressLine<< " "<< address->city<< " "<<address->state<<endl;
     }
  };
int main(void) {
   Address a1= Address("C-146, Sec-15","Noida","UP");
   Employee e1 = Employee(101,"Nakul",&a1);
          e1.display();
   return 0;
}
```

Output:

```
101 Nakul C-146, Sec-15 Noida UP
```

C++ Polymorphism

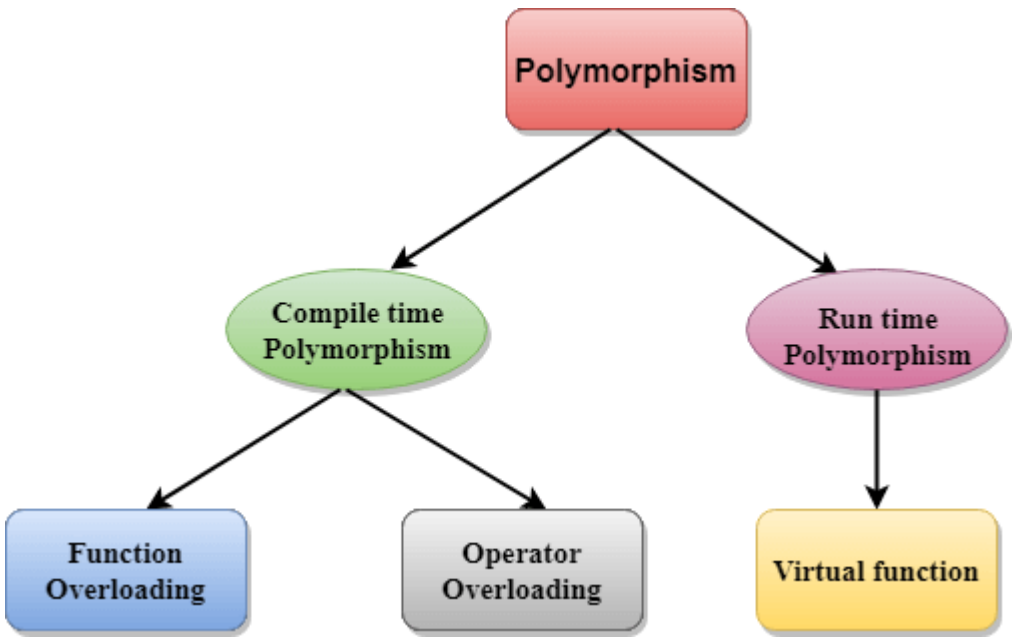
C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



- **Compile time polymorphism**: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
1.  class A                          //  base class declaration.
2.  {
3.      int a;
4.      public:
5.      void display()
6.      {
7.           cout<< "Class A ";
8.       }
9.  };
10. class B : public A                //  derived class declaration.
11. {
12.   int b;
13.   public:
14.  void display()
15.  {
16.      cout<<"Class B";
17.  }
18. };
```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism**: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but	Overriding is a run time polymorphism where more than one method is having the same name,

with the different number of parameters or the type of the parameters.	number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {        cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape {                    //  base class
   public:
virtual void draw(){             // virtual function
cout<<"drawing..."<<endl;
   }
};
class Rectangle: public Shape            //  inheriting Shape class.
{
 public:
 void draw()
   {
```

```
14.       cout<<"drawing rectangle..."<<endl;
15.    }
16. };
17. class Circle: public Shape                    //  inheriting Shape class.
18.
19. {
20.  public:
21.  void draw()
22.    {
23.       cout<<"drawing circle..."<<endl;
24.    }
25. };
26. int main(void) {
27.     Shape *s;                          //  base class pointer.
28.     Shape sh;                          // base class object.
29.        Rectangle rec;
30.         Circle cir;
31.       s=&sh;
32.      s->draw();
33.         s=&rec;
34.      s->draw();
35.     s=?
36.     s->draw();
37. }
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Animal {                          //  base class declaration.
4.      public:
5.      string color = "Black";
6.  };
7.  class Dog: public Animal                //  inheriting Animal class.
8.  {
9.   public:
10.     string color = "Grey";
11. };
12. int main(void) {
13.      Animal d= Dog();
14.     cout<<d.color;
15. }
```

Output:

```
Black
```

C++ Overloading (Function and Operator)

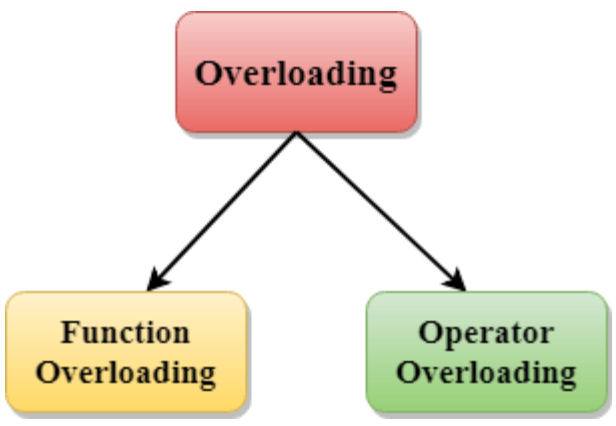
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                                  //    class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

Output:

```
30
55
```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```
1.  #include<iostream>
2.  using namespace std;
3.  int mul(int,int);
4.  float mul(float,int);
5.
6.
7.  int mul(int a,int b)
8.  {
9.      return a*b;
10. }
11. float mul(double x, int y)
12. {
13.     return x*y;
14. }
15. int main()
16. {
17.     int r1 = mul(6,7);
18.     float r2 = mul(0.2,3);
19.     std::cout << "r1 is : " <<r1<< std::endl;
20.     std::cout <<"r2 is : "  <<r2<< std::endl;
21.     return 0;
22. }
```

Output:

```
r1 is : 42
r2 is : 0.6
```

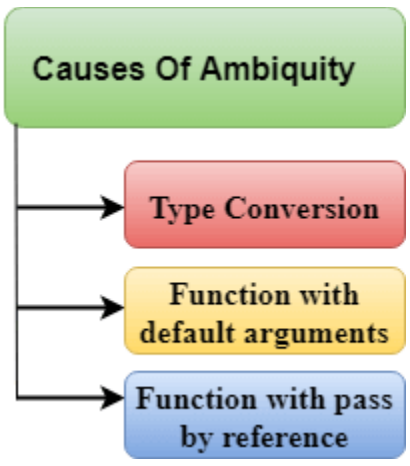
Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



- Type Conversion:

Let's see a simple example.

```
1.  #include<iostream>
2.  using namespace std;
3.  void fun(int);
4.  void fun(float);
5.  void fun(int i)
6.  {
7.      std::cout << "Value of i is : " <<i<< std::endl;
```

```
}
void fun(float j)
{
    std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

- Function with Default Arguments

Let's see a simple example.

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- Function with pass by reference

Let's see a simple example.

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
```

```
12. {
13. std::cout << "Value of x is : " <<x<< std::endl;
14. }
15. void fun(int &b)
16. {
17. std::cout << "Value of b is : " <<b<< std::endl;
18. }
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
1. #include <iostream>
2. using namespace std;
```

```
3.  class Test
4.  {
5.     private:
6.        int num;
7.     public:
8.        Test(): num(8){}
9.        void operator ++()        {
10.          num = num+2;
11.       }
12.       void Print() {
13.          cout<<"The Count is: "<<num;
14.       }
15. };
16. int main()
17. {
18.    Test tt;
19.    ++tt;  // calling of a function "void operator ++()"
20.    tt.Print();
21.    return 0;
22. }
```

Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
1.  #include <iostream>
2.  using namespace std;
3.  class A
4.  {
5.
6.     int x;
7.       public:
8.       A(){}
9.     A(int i)
10.    {
11.       x=i;
12.    }
13.    void operator+(A);
14.    void display();
15. };
16.
17. void A :: operator+(A a)
18. {
19.
20.    int m = x+a.x;
21.    cout<<"The result of the addition of two objects is : "<<m;
22.
23. }
24. int main()
25. {
26.    A a1(5);
27.    A a2(4);
28.    a1+a2;
29.    return 0;
30. }
```

Output:

```
The result of the addition of two objects is : 9
```

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5. void eat(){
6. cout<<"Eating...";
7.     }
8. };
9. class Dog: public Animal
10. {
11. public:
12. void eat()
13.     {
14.         cout<<"Eating bread...";
15.     }
16. };
17. int main(void) {
18.     Dog d = Dog();
19.     d.eat();
20.     return 0;
21. }
```

Output:

```
Eating bread...
```

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
   public:
   void display()
   {
       std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
       std::cout << "Value of y is : " <<y<< std::endl;
   }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;
}
```

Output:

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
```

```
5.   {
6.    cout << "Base class is invoked"<<endl;
7.   }
8.  };
9.  class B:public A
10. {
11.  public:
12.  void display()
13.  {
14.   cout << "Derived Class is invoked"<<endl;
15.  }
16. };
17. int main()
18. {
19.  A* a;    //pointer of base class
20.  B b;     //object of derived class
21.  a = &b;
22.  a->display();   //Late Binding occurs
23. }
```

Output:

```
Derived Class is invoked
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

1. **virtual void** display() = 0;

Let's see a simple example:

```
1.  #include <iostream>
2.  using namespace std;
3.  class Base
4.  {
5.     public:
6.     virtual void show() = 0;
7.  };
8.  class Derived : public Base
9.  {
10.    public:
11.    void show()
12.    {
13.        std::cout << "Derived class is derived from the base class." << std::endl;
14.    }
15. };
16. int main()
17. {
18.    Base *bptr;
```

```
19.    //Base b;
20.    Derived d;
21.    bptr = &d;
22.    bptr->show();
23.    return 0;
24. }
```

Output:

```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

C++ Abstraction

Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. **Abstract class**
2. **Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
1.  #include <iostream>
2.  using namespace std;
3.   class Shape
4.  {
5.     public:
6.     virtual void draw()=0;
7.  };
8.   class Rectangle : Shape
9.  {
10.    public:
11.     void draw()
12.    {
13.        cout < <"drawing rectangle..." < <endl;
14.    }
15. };
16. class Circle : Shape
17. {
18.    public:
19.     void draw()
20.    {
21.        cout <<"drawing circle..." < <endl;
22.    }
23. };
24. int main( ) {
```

```
25.     Rectangle rec;
26.     Circle cir;
27.     rec.draw();
28.     cir.draw();
29.    return 0;
30. }
```

Output:

```
drawing rectangle...
drawing circle...
```

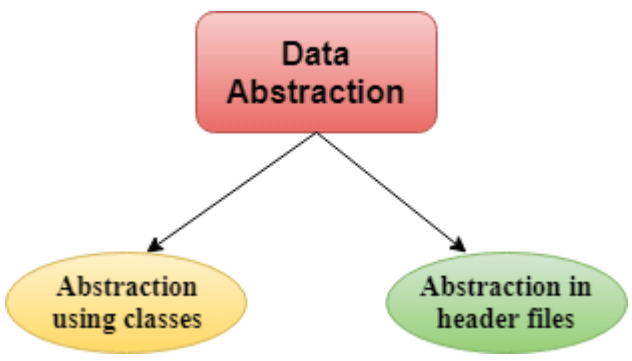
Data Abstraction in C++

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.



Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```
1. #include <iostream>
2. #include<math.h>
3. using namespace std;
4. int main()
5. {
```

```
6.  int n = 4;
7.    int power = 3;
8.    int result = pow(n,power);        // pow(n,power) is the  power function
9.    std::cout << "Cube of n is : " <<result<< std::endl;
10.   return 0;
11. }
```

Output:

```
Cube of n is : 64
```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

Let's see a simple example of data abstraction using classes.

```
1.  #include <iostream>
2.  using namespace std;
3.   class Sum
4.  {
5.  private: int x, y, z; // private variables
6.  public:
7.  void add()
8.  {
9.  cout<<"Enter two numbers: ";
10. cin>>x>>y;
11. z= x+y;
12. cout<<"Sum of two number is: "<<z<<endl;
13. }
14. };
15. int main()
16. {
17. Sum sm;
18. sm.add();
19. return 0;
20. }
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.