Lecture Notes

For

C++ Programming

By

LEARNERS TODAY, LEADERS TOMORROW

Basic Concepts of Object-Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming.

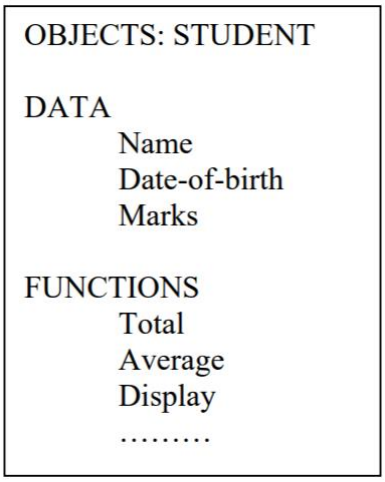
These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other's data or code. It is a sufficient to know the type of message accepted, and the type of response returned by the objects. Although different author represent them differently fig 1.5 shows two notations that are popularly used in object-oriented analysis and design.



Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types. For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language.

The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

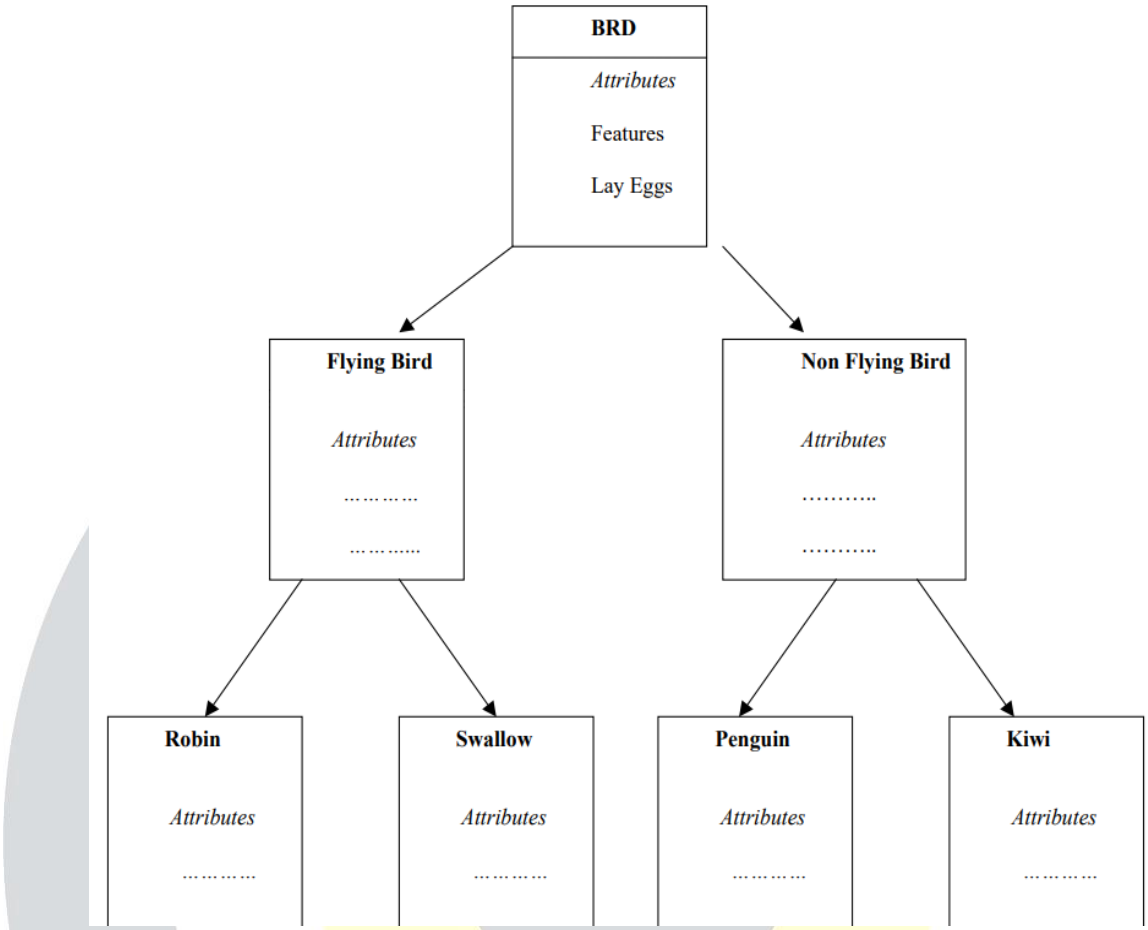
Fruit Mango;

Will create an object mango belonging to the class fruit.

Data Abstraction and Encapsulation

The wrapping up of data and function into a single unit (called class) is known as encapsulation. Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

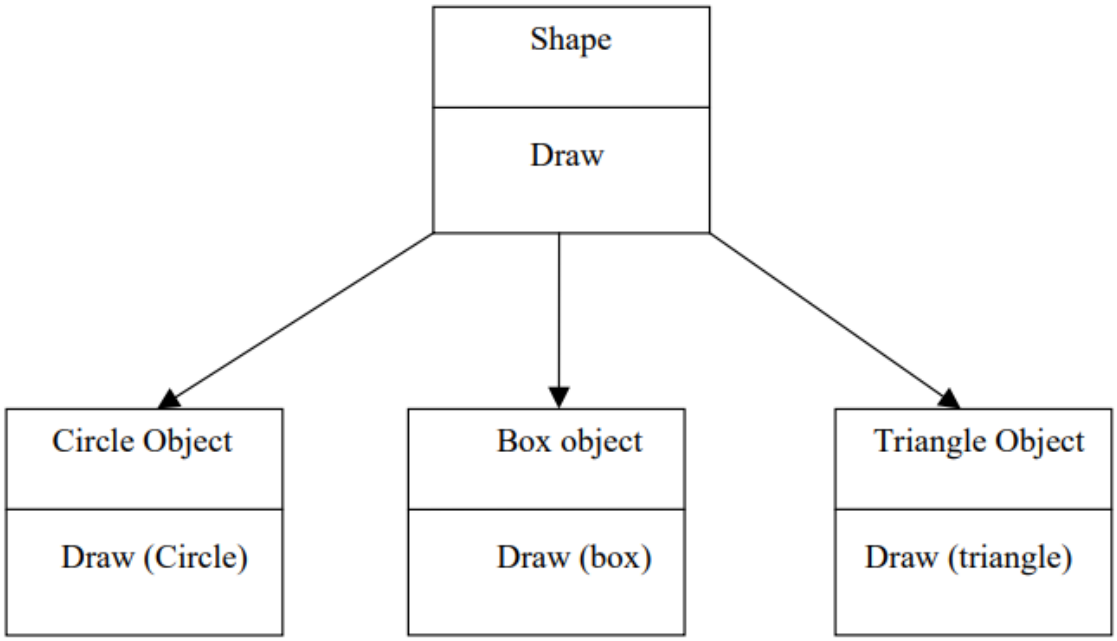
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created. The attributes are sometime called data members because they hold information. The functions that operate on these data are sometimes called methods or member function.



Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of hierarchical classification. For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in figure given below.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class i.e. almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of classes.



Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

Figure given below, illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as function overloading.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure “draw” in figure by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message Passing

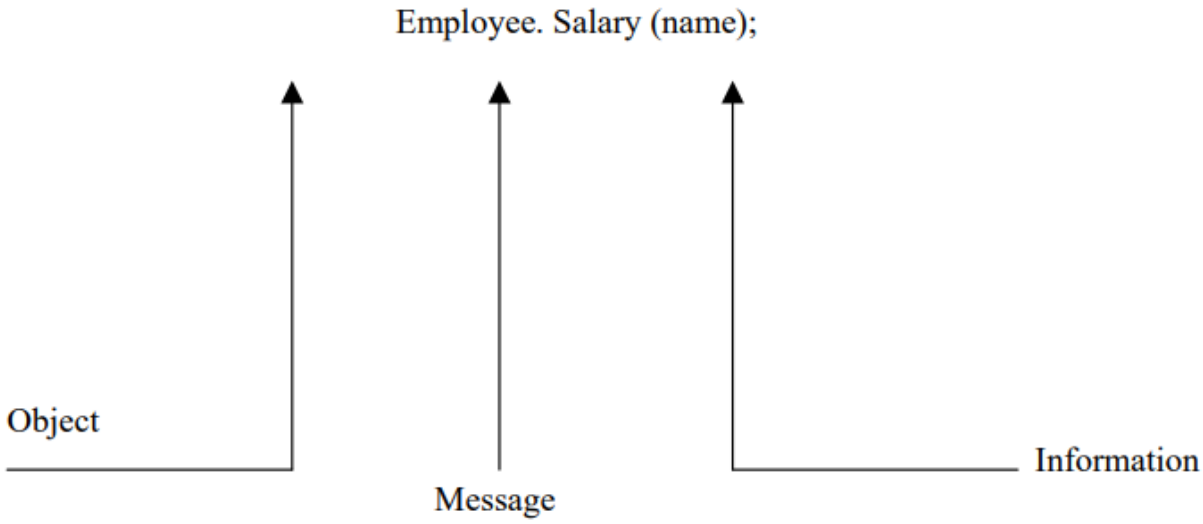
An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

- Creating classes that define object and their behavior,
- Creating objects from class definitions, and
- Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results.

Message passing involves specifying the name of object, the name of the function (message) and the information to be sent. Example:



Object has a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object Orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that cannot be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

C++ Program

Before starting the abcd of C++ language, you need to learn how to write, compile and run the first C++ program.

To write the first C++ program, open the C++ console and write the following code:

```
#include <iostream.h>
#include<conio.h>
void main() {
  clrscr();
  cout << "Welcome to C++ Programming.";
  getch();
}
```

#include<iostream.h> includes the standard input output library functions. It provides cin and cout methods for reading from input and writing to output respectively.

#include <conio.h> includes the console input output library functions. The getch() function is defined in conio.h file.

void main() The main() function is the entry point of every program in C++ language. The void keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is used to print the data "Welcome to C++ Programming." on the console.

getch() The getch() function asks for a single character. Until you press any key, it blocks the screen.

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.

```
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.

```
class Student
 {
    public:
    int id;  //field or data member
    float salary; //field or data member
    String name;//field or data member
 }
```

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.

```
#include <iostream>
using namespace std;
class Student {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
   Student s1; //creating an object of Student
   s1.id = 201;
   s1.name = "Sonoo Jaiswal";
   cout<<s1.id<<endl;
   cout<<s1.name<<endl;
   return 0;
}
```

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.

```
#include <iostream>
```

```
using namespace std;
class Student {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     void insert(int i, string n)
      {
         id = i;
         name = n;
      }
     void display()
      {
         cout<<id<<"  "<<name<<endl;
      }
};
int main(void) {
   Student s1; //creating an object of Student
   Student s2; //creating an object of Student
   s1.insert(201, "Sonoo");
   s2.insert(202, "Nakul");
   s1.display();
   s2.display();
   return 0;
}
```

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.

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     float salary;
     void insert(int i, string n, float s)
      {
         id = i;
         name = n;
         salary = s;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
   Employee e1; //creating an object of Employee
   Employee e2; //creating an object of Employee
   e1.insert(201, "Sonoo",99000);
   e2.insert(202, "Nakul", 29000);
   e1.display();
```

```
    e2.display();
    return 0;
}
```

Output:

```
201  Sonoo  99000
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
- Copy 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>
using namespace std;
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>
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:

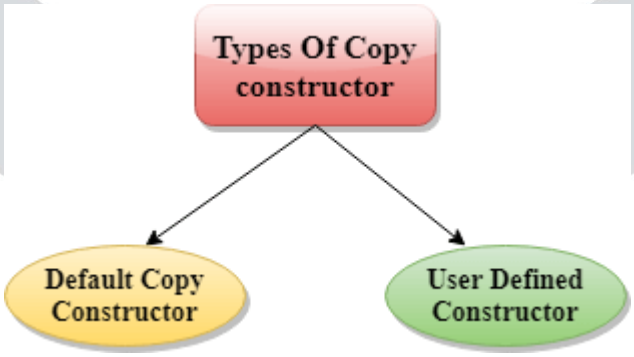
```
101  Sonoo  89000
102  Nakul  59000
```

C++ Copy Constructor

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
- **User Defined constructor:** The programmer defines the user-defined constructor.



Syntax Of User-defined Copy Constructor:

```
Class_name(const class_name &old_object);
```

Consider the following situation:

```
class A
{
   A(A &x) //  copy constructor.
  {
     // copyconstructor.
  }
}
```

In the above case, **copy constructor can be called in the following ways:**

- **A a2(a1);**
- **A a2 = a1;**

} **a1 initialises the a2 object.**

Let's see a simple example of the copy constructor.

// program of the copy constructor.

```
#include <iostream>
using namespace std;
class A
{
  public:
   int x;
   A(int a)            // parameterized constructor.
   {
    x=a;
   }
   A(A &i)             // copy constructor
   {
```

```
        x = i.x;
    }
};
int main()
{
  A a1(20);            // Calling the parameterized constructor.
 A a2(a1);             //  Calling the copy constructor.
 cout<<a2.x;
  return 0;
}
```

Output:

20

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

- When we initialize the object with another existing object of the same class type. For example, Student s1 = s2, where Student is the class.
- When the object of the same class type is passed by value as an argument.
- When the function returns the object of the same class type by value.

Two types of copies are produced by the constructor:

- Shallow copy
- Deep copy

Shallow Copy

- The default copy constructor can only produce the shallow copy.
- A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.

Let's understand this through a simple example:

```
#include <iostream>
```

```
using namespace std;

class Demo
{
   int a;
   int b;
   int *p;
   public:
   Demo()
   {
      p=new int;
   }
   void setdata(int x,int y,int z)
   {
      a=x;
      b=y;
      *p=z;
   }
   void showdata()
   {
      std::cout << "value of a is : " <<a<< std::endl;
      std::cout << "value of b is : " <<b<< std::endl;
      std::cout << "value of *p is : " <<*p<< std::endl;
   }
};
int main()
{
```

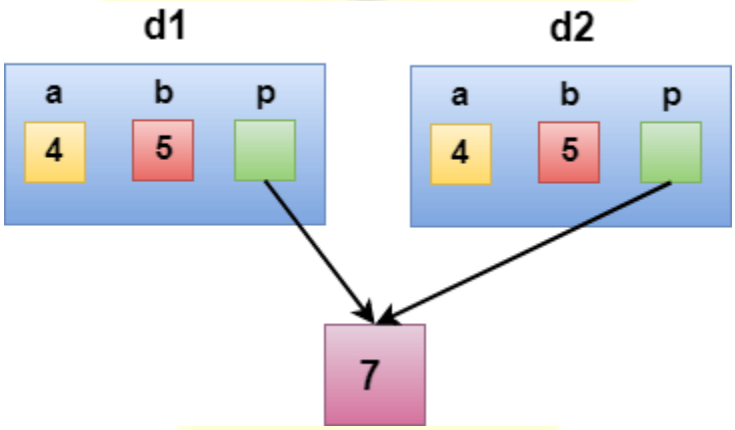
```
    Demo d1;
    d1.setdata(4,5,7);
    Demo d2 = d1;
    d2.showdata();
      return 0;
}
```

Output:

value of a is : 4

value of b is : 5

value of *p is : 7



In the above case, a programmer has not defined any constructor, therefore, the statement **Demo d2 = d1;** calls the default constructor defined by the compiler. The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer p of both the objects point to the same memory location. Therefore, when the memory of a field is freed, the memory of another field is also automatically freed as both the fields point to the same memory location. This problem is solved by the **user-defined constructor** that creates the **Deep copy**.

Deep copy

Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations. In this way, both the source and copy are

distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

Let's understand this through a simple example.

```
#include <iostream>
using namespace std;
class Demo
{
   public:
   int a;
   int b;
   int *p;

   Demo()
   {
      p=new int;
   }
   Demo(Demo &d)
   {
      a = d.a;
      b = d.b;
      p = new int;
      *p = *(d.p);
   }
   void setdata(int x,int y,int z)
   {
      a=x;
      b=y;
      *p=z;
```

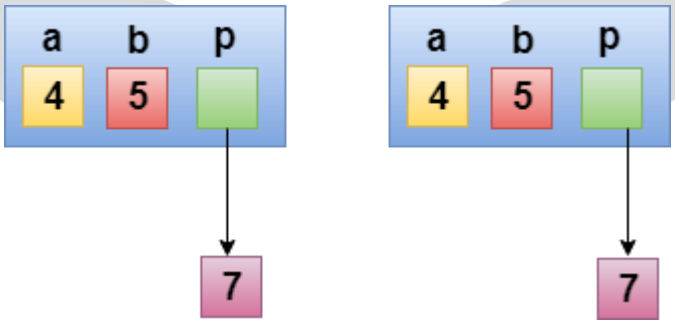
```
    }
    void showdata()
    {
        std::cout << "value of a is : " <<a<< std::endl;
        std::cout << "value of b is : " <<b<< std::endl;
        std::cout << "value of *p is : " <<*p<< std::endl;
    }
};
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
  Demo d2 = d1;
  d2.showdata();
  return 0;
}
```

Output:

value of a is : 4

value of b is : 5

value of *p is : 7



In the above case, a programmer has defined its own constructor, therefore the statement **Demo d2 = d1;** calls the copy constructor defined by the user. It creates the exact copy of the value

types data and the object pointed by the pointer p. Deep copy does not create the copy of a reference type variable.

Differences b/w Copy constructor and Assignment operator(=)

Copy Constructor	Assignment Operator
It is an overloaded constructor.	It is a bitwise operator.
It initializes the new object with the existing object.	It assigns the value of one object to another object.

Syntax of copy constructor:
Class_name(const class_name &object_name)
{
// body of the constructor.
}
	Syntax of Assignment operator:
	Class_name a,b;
	b = a;

○ The **copy constructor** is invoked when the new object is initialized with the existing object. ○ The object is passed as an argument to the function. ○ It returns the object.	The **assignment operator** is invoked when we assign the existing object to a new object.
Both the existing object and new object shares the different memory locations.	Both the existing object and new object shares the same memory location.
If a programmer does not define the copy constructor, the compiler will automatically generate the implicit default copy constructor.	If we do not overload the "=" operator, the bitwise copy will occur.

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.

```
#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;
}
```

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.

```
#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 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++ 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++.

```
#include <iostream>
using namespace std;
class Account {
  public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
```

```
    {
        this->accno = accno;
       this->name = name;
    }
    void display()
    {
       cout<<accno<<" "<<name<< " "<<rateOfInterest<<endl;
    }
};
float Account::rateOfInterest=6.5;
int main(void) {
  Account a1 =Account(201, "Sanjay"); //creating an object of Employee
  Account a2=Account(202, "Nakul"); //creating an object of Employee
  a1.display();
  a2.display();
  return 0;
}
```

Output:

201 Sanjay 6.5

202 Nakul 6.5

C++ Structure

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

```
struct structure_name
{
    // member declarations.
}
```

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:**

```
struct Student
{
   char name[20];
    int id;
    int age;
}
```

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:

```
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.

```
#include <iostream>
using namespace std;
 struct Rectangle
{
   int width, height;

 };
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
   cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
 return 0;
}
```

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.

```
#include <iostream>
using namespace std;
 struct Rectangle    {
  int width, height;
 Rectangle(int w, int h)
  {
     width = w;
     height = h;
  }
  void areaOfRectangle() {
   cout<<"Area of Rectangle is: "<<(width*height); }
 };
int main(void) {
   struct Rectangle rec=Rectangle(4,6);
   rec.areaOfRectangle();
  return 0;
}
```

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.

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
int main()
{
   week day;
   day = Friday;
   cout << "Day: " << day+1<<endl;
   return 0;
}
```

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++

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

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.

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

```
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output:

Length of box: 10

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

```
#include <iostream>
using namespace std;
class B;          // forward declarartion.
class A
{
```

```
    int x;
    public:
    void setdata(int i)
    {
       x=i;
    }
    friend void min(A,B);        // friend function.
};
class B
{
    int y;
    public:
    void setdata(int i)
    {
       y=i;
    }
    friend void min(A,B);                // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
    std::cout << a.x << std::endl;
    else
    std::cout << b.y << std::endl;
}
  int main()
{
  A a;
```

```
  B b;
  a.setdata(10);
  b.setdata(20);
  min(a,b);
   return 0;
}
```

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++ Math Functions

C++ offers some basic math functions and the required header file to use these functions is <math.h>

Trigonometric functions

Method	Description
cos(x)	It computes the cosine of x.
sin(x)	It computes the sine of x.
tan(x)	It computes the tangent of x.
acos(x)	It finds the inverse cosine of x.
asin(x)	It finds the inverse sine of x.
atan(x)	It finds the inverse tangent of x.
atan2(x,y)	It finds the inverse tangent of a coordinate x and y.

Hyperbolic functions

Method	Description
cosh(x)	It computes the hyperbolic cosine of x.
sinh(x)	It computes the hyperbolic sine of x.
tanh(x)	It computes the hyperbolic tangent of x.

acosh(x)	It finds the arc hyperbolic cosine of x.
asinh(x)	It finds the arc hyperbolic sine of x.
atanh(x)	It finds the arc hyperbolic tangent of x.

Exponential functions

Method	Description
exp(x)	It computes the exponential e raised to the power x.
frexp(value_type x,int* exp)	It breaks a number into significand and 2 raised to the power exponent.
ldexp(float x, int e)	It computes the product of x and 2 raised to the power e.
log(x)	It computes the natural logarithm of x.
log10(x)	It computes the common logarithm of x.
modf()	It breaks a number into an integer and fractional part.
exp2(x)	It computes the base 2 exponential of x.
expm1(x)	It computes the exponential raised to the power x minus one.
log1p(x)	It computes the natural logarithm of x plus one.
log2(x)	It computes the base 2 logarithm of x.
logb(x)	It computes the logarithm of x.

scalbn(x, n)	It computes the product of x and FLT_RADX raised to the power n.
scalbln(x, n)	It computes the product of x and FLT_RADX raised to the power n.
ilogb(x)	It returns the exponent part of x.

Floating point manipulation functions

Method	Description
copysign(x,y)	It returns the magnitude of x with the sign of y.
nextafter(x,y)	It represents the next representable value of x in the direction of y.
nexttoward(x,y)	It represents the next representable value of x in the direction of y.

Maximum, Minimum and Difference functions

Method	Description
fdim(x,y)	It calculates the positive difference between x and y.
fmax(x,y)	It returns the larger number among two numbers x and y.
fmin()	It returns the smaller number among two numbers x and y .

Power functions

Method	Description
pow(x,y)	It computes x raised to the power y.
sqrt(x)	It computes the square root of x.
cbrt(x)	It computes the cube root of x.
hypot(x,y)	It finds the hypotenuse of a right angled triangle.

Nearest integer operations

Method	Description
ceil(x)	It rounds up the value of x.
floor(x)	It rounds down the value of x.
round(x)	It rounds off the value of x.
lround(x)	It rounds off the value of x and cast to long integer.
llround(x)	It rounds off the value of x and cast to long long integer.
fmod(n,d)	It computes the remainder of division n/d.
trunc(x)	It rounds off the value x towards zero.
rint(x)	It rounds off the value of x using rounding mode.
lrint(x)	It rounds off the value of x using rounding mode and cast to long integer.
llrint(x)	It rounds off the value x and cast to long long integer.

nearbyint(x)	It rounds off the value x to a nearby integral value.
remainder(n,d)	It computes the remainder of n/d.
remquo()	It computes remainder and quotient both.

Other functions

Method	Description
fabs(x)	It computes the absolute value of x.
abs(x)	It computes the absolute value of x.
fma(x,y,z)	It computes the expression x*y+z.

Macro functions

Method	Description
fpclassify(x)	It returns the value of type that matches one of the macro constants.
isfinite(x)	It checks whether x is finite or not.
isinf()	It checks whether x is infinite or not.
isnan()	It checks whether x is nan or not.
isnormal(x)	It checks whether x is normal or not.
signbit(x)	It checks whether the sign of x is negative or not.

Comparison macro functions

Method	Description
isgreater(x,y)	It determines whether x is greater than y or not.
isgreaterequal(x,y)	It determines whether x is greater than or equal to y or not.
less(x,y)	It determines whether x is less than y or not.
islessequal(x,y)	It determines whether x is less than or equal to y.
islessgreater(x,y)	It determines whether x is less or greater than y or not.
isunordered(x,y)	It checks whether x can be meaningfully compared or not.

Error and gamma functions

Method	Description
erf(x)	It computes the error function value of x.
erfc(x)	It computes the complementary error function value of x.
tgamma(x)	It computes the gamma function value of x.
lgamma(x)	It computes the logarithm of a gamma function of x.

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.

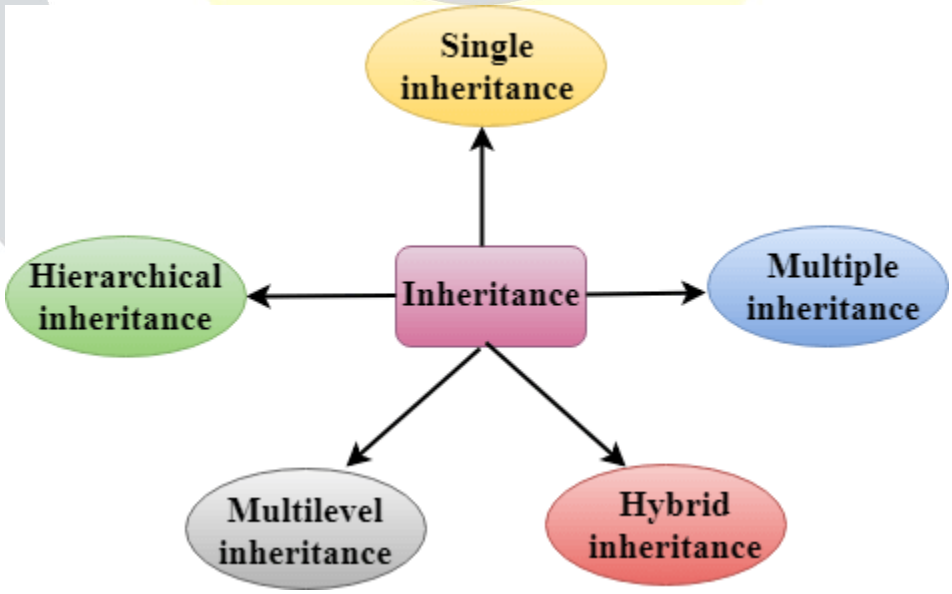
Advantage of C++ Inheritance

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.

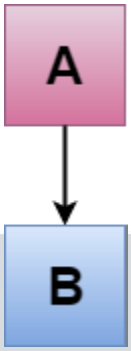
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- 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() {
    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

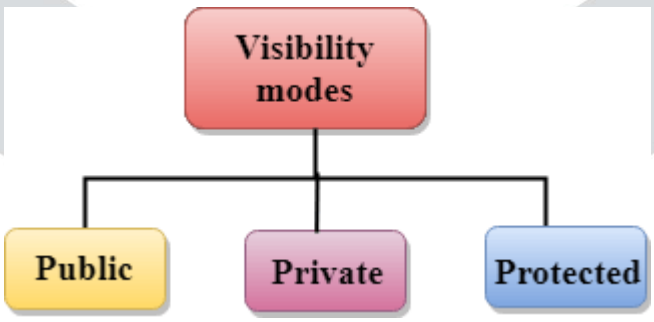
In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

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:

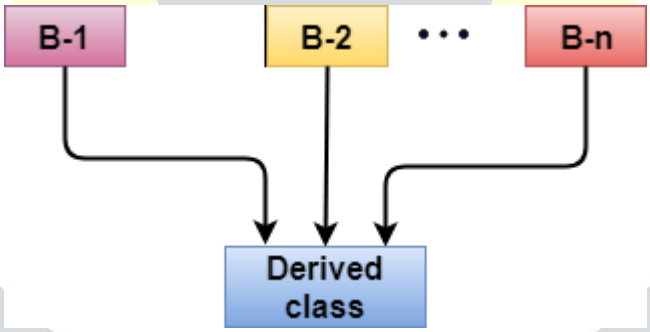
Eating...

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.

Ambiguity 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()
   {
      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.

  }
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

```
class A
{
  public:
void display()
{
  cout<<?Class A?;
}
} ;
```

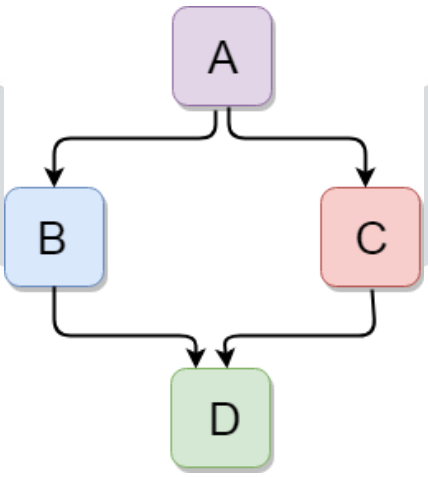
```
class B
{
 public:
 void display()
{
 cout<<?Class B?;
}
} ;
```

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.

```
int main()
{
   B b;
   b.display();              // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

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

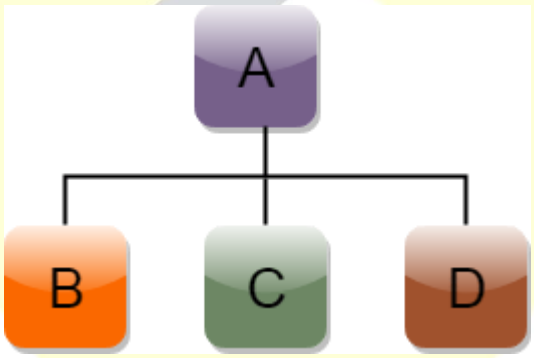
Enter the value of c is :

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.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
```

```
    }
};
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

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

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
```

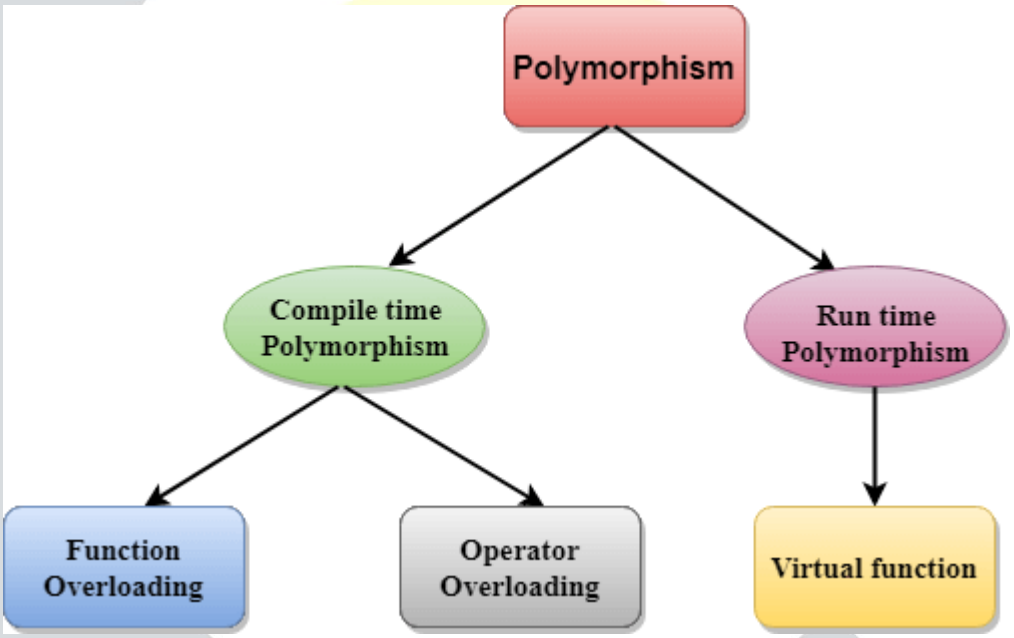
C++ Polymorphism

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.

```
class A                          //  base class declaration.
    {
        int a;
        public:
        void display()
        {
             cout<< "Class A ";
         }
    };
   class B : public A                    //  derived class declaration.
  {
     int b;
     public:
     void display()
    {
         cout<<"Class B";
    }
  };
```

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 with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, 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()
{
cout<<"drawing rectangle..."<<endl;
}
};
class Circle: public Shape               //  inheriting Shape class.

{
public:
void draw()
{
cout<<"drawing circle..."<<endl;
}
};
int main() {
Shape *s;                       //  base class pointer.
Shape sh;                       // base class object.
Rectangle rec;
Circle cir;
s=&sh;
```

```
s->draw();
s=&rec;
s->draw();
s=?
s->draw();
}
```

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.

```
#include <iostream>
using namespace std;
class Animal {                          //  base class declaration.
   public:
   string color = "Black";
};
class Dog: public Animal                // inheriting Animal class.
{
 public:
   string color = "Grey";
};
int main(void) {
```

```
    Animal d= Dog();
    cout<<d.color;
}
```

Output:

Black

C++ Overloading (Function and Operator)

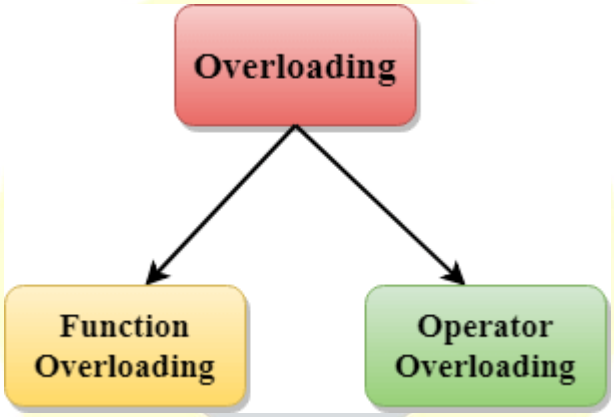
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ 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.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);
int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
   return 0;
}
```

Output:

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

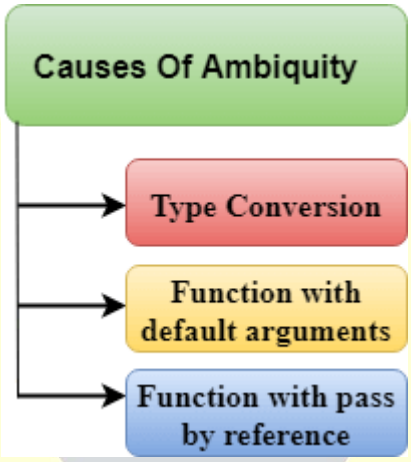
Function Overloading and Ambiguity

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.



C++ Overloading

Type Conversion:

Let's see a simple example.

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   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)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
```

```
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

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 ++.

#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()        {
        num = num+2;
     }
```

```
    void Print() {
        cout<<"The Count is: "<<num;
    }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output:

The Count is: 10

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

// program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
{

   int x;
     public:
     A(){}
   A(int i)
   {
      x=i;
```

```
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{
    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;
}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

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.

```
#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++ 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()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
```

```
 a = &b;
 a->display();   //Late Binding occurs
}
```

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:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
```

```
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

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++ 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.

```
#include <iostream>
using namespace std;
 class Shape
{
   public:
   virtual void draw()=0;
};
 class Rectangle : Shape
{
   public:
    void draw()
   {
      cout < <"drawing rectangle..." < <endl;
   }
};
class Circle : Shape
{
   public:
    void draw()
```

```
    {
        cout <<"drawing circle..." < <endl;
    }
};
int main( ) {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
   return 0;
}
```

Output:

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

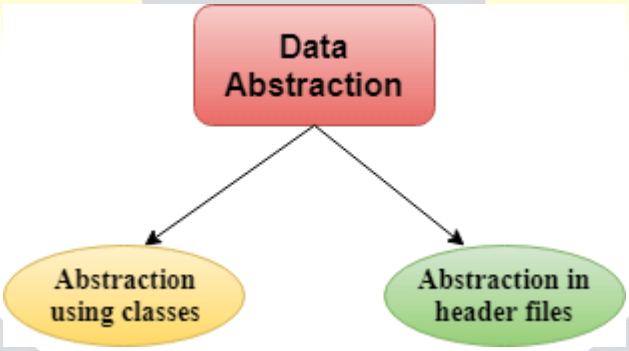
C++ Data Abstraction

- 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:

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.

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
int n = 4;
int power = 3;
int result = pow(n,power);        // pow(n,power) is the  power function
std::cout << "Cube of n is : " <<result<< std::endl;
return 0;
}
```

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.

```
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variables
public:
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()
{
Sum sm;
sm.add();
return 0;
}
```

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.

C++ Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

C++ String Example

Let's see the simple example of C++ string.

```
#include <iostream>
using namespace std;
int main( ) {
   string s1 = "Hello";
      char ch[] = { 'C', '+', '+'};
      string s2 = string(ch);
      cout<<s1<<endl;
      cout<<s2<<endl;
}
```

Output:

```
Hello
C++
```

C++ String Length Example

Let's see the simple example of finding the string length using strlen() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
```

```
    char ary[] = "Welcome to C++ Programming";
    cout << "Length of String = " << strlen(ary)<<endl;
    return 0;
}
```

Output:

Length of String = 26

C++ String Functions

C++ String compare()

This function compares the value of the string object to the sequence of characters specified by its parameter.

Syntax :

Suppose str1 and str2 are two strings and we want to compare these two strings then its syntax would look like:

int k= str1.compare(str2);

k==0 : If k contains value zero, it means both the strings are equal.

k!=0 : If k does contain value zero, it means both the strings are unequal.

k>0 : If k contains value more than zero, either the value of the first character is greater in the compared string or all the compared characters match but the compared string is longer.

k<0 : If k contains value less than zero, either the value of the first character is lower in the compared string or all the compared characters match but the compared string is shorter.

Example 1

```
#include<iostream>
using namespace std;
void main()
```

```
{
  string str1="Hello";
  string str2="claritech";
   int k= str1.compare(str2);
   if(k==0)
       cout<<"Both the strings are equal";
    else
      cout<<"Both the strings are unequal";
}
```

Output:

Both the strings are unequal

C++ String length()

This function is used to find the length of the string in terms of bytes.

This is the actual number of bytes that conform the contents of the string , which is not necessarily equal to the storage capacity.

Syntax

```
int len = s1.length();
```

Parameters

This function contains single parameter.

Return Value

This function returns the integer value in terms of bytes.

Here, s1 is a string and strlen function calculates the length of the string s1 and stores its value in len variable.

```
#include<iostream>
using  namespace std;
int main()
{
string s1 = "Welcome to claritech";
int len = s1.length();
cout<< "length of the string is : " << len;
return 0;
}
```

Output:

Length of the string is 21

Consider a string s1 and s1 contains value i.e Welcome to claritech .We calculate the length of the string using strlen function and stores in variable len.

C++ String swap()

This function is used to exchange the values of two string objects.

Syntax

Consider two strings s1 and s2 , we want to exchange the values of these two string objects. Its syntax would be :

s1.swap(s2)

Parameters

It contains single parameter, whose value is to be exchanged with that of the string object.

Return value

It does not return any value.

```
#include<iostream>
using namespace std;
int  main()
{
string r = "10";
string m = "20"
cout<<"Before  swap r contains " << r <<"rupees"<<'\n';
cout<<"Before  swap m contains " << m <<"rupees"<<'\n';
r.swap(m);
cout<< "After  swap r contains " << r<<"rupees"<<'\n';
cout<< "After  swap m contains " << m<<"rupees";
return 0;
}
```

Output:

Before swap r contains 10 rupees

Before swap m contains 20 rupees

After swap r contains 20 rupees

After swap m contains 10 rupees

In this example, r and m are two string objects containing 10 and 20 rupees respectively. We swap their values by using swap function. After swapping, r contains 20 rupees and m contains 10 rupees.

C++ String substr()

```
// CPP program to illustrate substr()
#include <string.h>
#include <iostream>
using namespace std;
```

```
int main()
{
    // Take any string
    string s1 = "Geeks";

    // Copy three characters of s1 (starting
    // from position 1)
    string r = s1.substr(1, 3);

    // prints the result
    cout << "String is: " << r;
     return 0;
}
```

Output:

String is: eek

C++ String size()

This function is used to return the length of the string in terms of bytes. It defines the actual number of bytes that conform to the contents of the string object which is not necessarily equal to the capacity.

Syntax

Consider a string object named as 'str'. To calculate the size of this string object, its syntax would be :

str.size()

Parameters

This function does not contain any parameter.

Return Value

This function returns the number of characters present in the string object.

Example 1

```
#include<iostream>
using namespace std;
int main()
{
string str ="Welcome to the claritech tutorial";
int size = str.size();
cout << "length of the string is :" <<size;
return 0;
}
```

Output:

length of the string is : 34

In this example, str is a string object containing value "Welcome to the claritech tutorial". We calculate the length of the string using 'size' function.

C++ String replace()

This function replaces the portion of string that begins at character position pos and spans len characters.

Syntax

Consider two strings str1 and str2. Syntax would be:

str1.replace(pos,len,str2);

Parameters

str : str is a string object, whose value to be copied in another string object.

pos : pos defines the position, whose character to be replaced.

len : Number of characters to be replaced by another string object.

subpos : It defines the position of the first character of string object that is to be copied to another object as replacement.

sublen : Number of characters of string object to be copied into another string object.

n : Number of characters to be copied into an another string object.

Return value

This function does not return any value.

Example 1

First example shows how to replace given string by using position and length as parameters.

```
#include<iostream>
using namespace std;
int  main()
{
string str1 = "This is C language";
string str2 = "C++";
cout << "Before replacement, string is :"<<str1<<'\n';
str1.replace(8,1,str2);
cout << "After replacement, string is :"<<str1<<'\n';
return 0;
}
```

Output:

Before replacement , string is This is C language

After replacement, string is This is C++ language

Example 2

Second example shows how to replace given string using position and length of the string which is to be copied in another string object.

```
#include<iostream>

using namespace std;

int main()

{

string str1 ="This is C language"

 string str3= "java language";

cout <<"Before replacement, String is "<<str1<<'\n';

str1.replace(8,1,str3,0,4);

cout<<"After  replacement,String is "<<str1<<'\n';

return 0;

}
```

Output:

Before replacement, String is This is C language

After replacement, String is This is java language

Example 3

Third example shows how to replace the string by using string and number of characters to be copied as parameters.

```
#include<iostream>

using namespace std;
```

```
int main()
{
string str1="This is C language";
cout<<"Before replacement,string is"<<str1<<'\n';
str1.replace(8,1,"C##",2);
cout<<"After replacement,string is"<<str1;
return 0;
}
```

Output:

Before replacement,string is This is C language

After replacement,string is This is C# language

C++ String append()

This function is used to extend the string by appending at the end of the current value.

Syntax

Consider string str1 and str2. Syntax would be :

Str1.append(str2);

Str1.append(str2,pos,len);

Str1.append(str2,n);

Parameters

str : String object which is to be appended in another string object.

pos : It determines the position of the first character that is to be appended to another object.

len : Number of characters to be copied in another string object as substring.

n : Number of characters to copy.

Return value

This function does not return any value.

Example 1

Let's see the example of appending the string in another string object.

```
#include<iostream>
using namespace std;
int main()
{
string str1="Welcome to C++ programming";
string str2="language";
cout<<"Before appending,string value is"<<str1<<'\n';
str1.append(str2);
cout<<"After appending, string value is"<<str1<<'\n';
return 0;
}
```

Output:

Before appending,string value is Welcome to C++ programming

After appending,string value is Welcome to C++ programming language

Example 2

Let's see the example of appending the string by using position and length as parameters.

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

```
int main()
{
 string str1 = "Mango is my favourite" ;
string str2 ="fruit";
cout<<"Before appending, string value is :" <<str1<<'\n';
str1.append(str2,0,5);
cout<<"After appending, string value is :" <<str1<<'\n';
return 0;
}
```

Output:

Before appending, string value is Mango is my favourite

After appending, string value is Mango is my favourite fruit

Example 3

Let's see another example.

```
#include<iostream>
using namespace std;
int main()
{
 string str1 = "Kashmir is nature";
str1.append("of  beauty",9) ;
cout<<"String value is :"<<str1;
return 0;
}
```

Output:

String value is Kashmir is nature of beauty

C++ String at()

This function is used for accessing individual characters.

Syntax

Consider a string str. To find the position of a particular character, its syntax would be

str.at(pos);

Parameters

pos : It defines the position of the character within the string.

Return value

It returns the character specified at that position.

```
#include<iostream>
using namespace std;
int main()
{
string str = "Welcome to claritech tutorial";
cout<<"String contains :";
for(int i=0; i<str.length(); i++)
{
cout<<  str.at(i);
}
cout<<'\n';
cout<<"String is shown again";
```

```
for(int j=0 ; j<str.length(); j++)
{
cout<< str[j];
}
return 0;
}
```

Output:

String contains Welcome to claritech tutorial

C++ String Find()

This function is used for finding a specified substring.

Syntax

Consider two strings str1 and str2. Syntax would be :

```
str1.find(str2);
```

Parameters

str : String to be searched for.

pos : It defines the position of the character at which to start the search.

n : Number of characters in a string to be searched for.

ch : It defines the character to search for.

Return value

It returns the position of the first character of first match.

Example 1

Let's see the simple example.

```
#include<iostream>
using namespace std;
int main()
{
string str= "java is the best programming language";
cout <<  str<<'\n';
cout <<" Position of the programming word is :";
cout<< str.find("programming");
return 0;
}
```

Output:

Java is the best programming language

Position of the programming word is 17

Example 2

Let's see simple example by passing position of a character as a parameter.

```
#include<iostream>
using namespace std;
int main()
{
string str= "Mango is my favorite fruit";
cout <<  str<<'\n';
cout<< " position of fruit is :";
cout<< str.find("fruit",12);
return 0;
}
```

Output:

Mango is my favorite fruit

Position of fruit is 21

Example 3

Let's see simple example of finding a single character.

```
#include<iostream>
using  namespace std;
int main()
{
string str = "claritech";
cout << "String contains :" << str;
cout<< "position of t is :" << str.find('t');
return 0;
}
```

Output:

String contains : claritech

Position of t is 5

C++ String insert()

This function is used to insert a new character, before the character indicated by the position pos.

Syntax

Consider two strings str1 and str2, pos is the position.

Syntax would be :

```
str1.insert(pos,str2);
```

Parameters

str : String object to be inserted in another string object.

pos : It defines the position at which new content is inserted just before the specified position.

subpos : It defines the position of first character in string str which is to be inserted in another string object.

sublen : It defines the number of characters of string str to be inserted in another string object.

n : It determines the number of characters to be inserted.

c : Character value to insert.

Example 1

Let's see the simple example.

```
#include<iostream>
using namespace std;
int main()
string str1= "C tutorial";
cout<<"String contains :" <<str1<<'\n';
cout<<"After insertion, String value is :"<<str1.insert(1,"++");
return 0;
}
```

Output:

String contains : C tutorial

After insertion, String value is C++ tutorial

Example 2

Let's the simple example of insertion when subpos and sublen are given.

```
#include<iostream>
using namespace std;
int main()
{
string str1 = "C++  is a language";
string str2 = "programming";
cout<<"String contains :" <<str1<<'\n';
cout<<"After insertion, String is :"<< str1.insert(9,str2,0,11);
return 0;
}
```

Output:

String contains C++ is a language

After insertion, String is C++ is a programming language

Example 3

Let's see the simple example of insertion when number of characters to be inserted are given.

```
#include<iostream>
using namespace std;
int main()
{
string str = "Maths is favorite subject" ;
cout<<"String contains :"<<str<<'\n';
cout<<"After insertion, String contains :<<str.insert(9,"my",2);
return 0;
}
```

Output:

String contains : Maths is favorite subject

After insertion, String contains : Maths is my favorite subject

C++ String push_back()

This function is used to add new character ch at the end of the string, increasing its length by one.

Syntax

Consider a string s1 and character ch . Syntax would be :

s1.push_back(ch);

Parameters

ch : New character which is to be added.

Return value

It does not return any value.

Example 1

Let's see simple example.

```
#include<iostream>
using namespace std;
    int main()
    {
    string s1 = "Hell";
    cout<< "String is :" <<s1<<'\n';
    s1.push_back('o');
```

```
 cout<<"Now, string is :"<<s1;
return 0;
 }
```

Example 2

Let's consider another simple example.

```
#include<iostream>
using namespace std;
int main()
{
string str = "java tutorial ";
cout<<"String contains :" <<str<<'\n';
str.push_back('1');
cout<<"Now,string is : "<<str;
return 0;
}
```

Output:

```
String contains :java tutorial
Now,string is java tutorial 1
```

Example 3

Let's see the example of inserting an element at the end of vector.

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

```
int main()
{
vector<char> s;
s.push_back('j');
s.push_back('a');
s.push_back('v');
s.push_back('a');
for(int i=0;i<s.size();i++)
cout<<s[i];
return 0;
}
```

Output:

Java

C++ String pop_back()

This function is used to remove a last character of a string, decreasing its length by one.

Syntax

Consider a string str. Syntax would be :

```
str.pop_back();
```

Parameters

This function does not contain any parameter.

Return value

This function does not return any value.

Example 1

Let's see the simple example.

```
#include<iostream>
using namespace std;
int main()
{
string str ="javac";
str.pop_back();
cout<<str;
return 0;
}
```

Output:

Java

C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

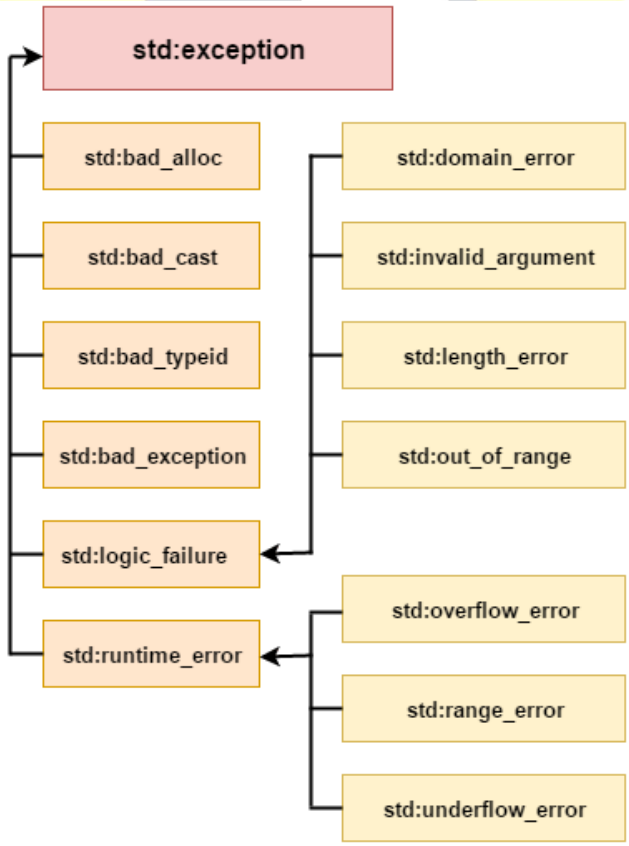
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

Exception	Description
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

Moreover, we can create user-defined exception which we will learn in next chapters.

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ try block is used to place the code that may occur exception. The catch block is used to handle the exception.

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
}
int main () {
  int i = 50;
  int j = 0;
  float k = 0;
    k = division(i, j);
    cout << k << endl;
  return 0;
}
```

Output:

Floating point exception (core dumped)

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
```

```
   if( y == 0 ) {
      throw "Attempted to divide by zero!";
   }
   return (x/y);
}
int main () {
   int i = 25;
   int j = 0;
   float k = 0;
   try {
      k = division(i, j);
      cout << k << endl;
   }catch (const char* e) {
      cerr << e << endl;
   }
   return 0;
}
```

Output:

Attempted to divide by zero!

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting exception class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which std::exception class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
   public:
      const char * what() const throw()
      {
         return "Attempted to divide by zero!\n";
      }
};
int main()
{
   try
   {
      int x, y;
      cout << "Enter the two numbers : \n";
      cin >> x >> y;
      if (y == 0)
      {
         MyException z;
```

```
            throw z;
        }
        else
        {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(exception& e)
    {
        cout << e.what();
    }
}
```

Output:

Enter the two numbers :

10

2

x / y = 5

Output:

Enter the two numbers :

10

0

Attempted to divide by zero!

C++ Files and Streams

In C++ programming we are using the **iostream** standard library, it provides **cin** and **cout** methods for reading from input and writing to output respectively.

To read and write from a file we are using the standard C++ library called **fstream**. Let us see the data types define in fstream library is:

Data Type	Description
fstream	It is used to create files, write information to files, and read information from files.
ifstream	It is used to read information from files.
ofstream	It is used to create files and write information to the files.

C++ FileStream example: writing to a file

Let's see the simple example of writing to a text file **testout.txt** using C++ FileStream programming.

```
#include <iostream>

#include <fstream>

using namespace std;

int main () {

  ofstream filestream("testout.txt");

  if (filestream.is_open())

  {

    filestream << "Welcome to javaTpoint.\n";

    filestream << "C++ Tutorial.\n";
```

```
  filestream.close();
 }
 else cout <<"File opening is fail.";
 return 0;
}
```

Output:

The content of a text file **testout.txt** is set with the data:

Welcome to javaTpoint.

C++ Tutorial.

C++ FileStream example: reading from a file

Let's see the simple example of reading from a text file **testout.txt** using C++ FileStream programming.

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 string srg;
 ifstream filestream("testout.txt");
 if (filestream.is_open())
 {
  while ( getline (filestream,srg) )
  {
   cout << srg <<endl;
  }
  filestream.close();
 }
```

```
  else {
     cout << "File opening is fail."<<endl;
    }
  return 0;
}
```

Note: Before running the code a text file named as **"testout.txt"** is need to be created and the content of a text file is given below:
Welcome to javaTpoint.
C++ Tutorial.

Output:

Welcome to javaTpoint.

C++ Tutorial.

C++ Read and Write Example

Let's see the simple example of writing the data to a text file **testout.txt** and then reading the data from the file using C++ FileStream programming.

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
   char input[75];
   ofstream os;
   os.open("testout.txt");
   cout <<"Writing to a text file:" << endl;
   cout << "Please Enter your name: ";
   cin.getline(input, 100);
   os << input << endl;
   cout << "Please Enter your age: ";
```

```
    cin >> input;
    cin.ignore();
    os << input << endl;
    os.close();
    ifstream is;
    string line;
    is.open("testout.txt");
    cout << "Reading from a text file:" << endl;
    while (getline (is,line))
    {
    cout << line << endl;
    }
    is.close();
    return 0;
}
```

Output:

```
Writing to a text file:
Please Enter your name: Nakul Jain
Please Enter your age: 22
Reading from a text file:   Nakul Jain
22
```

C++ getline()

The cin is an object which is used to take input from the user but does not allow to take the input in multiple lines. To accept the multiple lines, we use the getline() function. It is a pre-defined function defined in a <string.h> header file used to accept a line or a string from the input stream until the delimiting character is encountered.

Syntax of getline() function:

There are two ways of representing a function:

The first way of declaring is to pass three parameters.

istream& getline(istream& is, string& str, char delim);

The above syntax contains three parameters, i.e., is, str, and delim.

Where,

is: It is an object of the istream class that defines from where to read the input stream.

str: It is a string object in which string is stored.

delim: It is the delimiting character.

Return value

This function returns the input stream object, which is passed as a parameter to the function.

The second way of declaring is to pass two parameters.

istream& getline(istream& is, string& str);

The above syntax contains two parameters, i.e., is and str. This syntax is almost similar to the above syntax; the only difference is that it does not have any delimiting character.

Where,

is: It is an object of the istream class that defines from where to read the input stream.

str: It is a string object in which string is stored.

Return value

This function also returns the input stream, which is passed as a parameter to the function.

Let's understand through an example.

First, we will look at an example where we take the user input without using getline() function.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
string name; // variable declaration
std::cout << "Enter your name :" << std::endl;
cin>>name;
cout<<"\nHello "<<name;
return 0;
}
```

In the above code, we take the user input by using the statement cin>>name, i.e., we have not used the getline() function.

Output

Enter your name :

John Miller

Hello John

In the above output, we gave the name 'John Miller' as user input, but only 'John' was displayed. Therefore, we conclude that cin does not consider the character when the space character is encountered.

Let's resolve the above problem by using getline() function.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
string name; // variable declaration.
std::cout << "Enter your name :" << std::endl;
getline(cin,name); // implementing a getline() function
cout<<"\nHello "<<name;
return 0;}
```

In the above code, we have used the getline() function to accept the character even when the space character is encountered.

Output

```
Enter your name :
John Miller
Hello John Miller
```

In the above output, we can observe that both the words, i.e., John and Miller, are displayed, which means that the getline() function considers the character after the space character also.

When we do not want to read the character after space then we use the following code:

```
#include <iostream>
```

```
#include<string.h>
using namespace std;
int main()
{
string profile; // variable declaration
std::cout << "Enter your profile :" << std::endl;
getline(cin,profile,' '); // implementing getline() function with a delimiting character.
cout<<"\nProfile is :"<<profile;
}
```

In the above code, we take the user input by using getline() function, but this time we also add the delimiting character(') in a third parameter. Here, delimiting character is a space character, means the character that appears after space will not be considered.

Output

```
Enter your profile :
Software Developer
Profile is: Software
```

Getline Character Array

We can also define the getline() function for character array, but its syntax is different from the previous one.

Syntax

```
istream& getline(char* , int size);
```

In the above syntax, there are two parameters; one is char*, and the other is size.

Where,

char*: It is a character pointer that points to the array.

Size: It acts as a delimiter that defines the size of the array means input cannot cross this size.

Let's understand through an example.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
char fruits[50]; // array declaration
cout<< "Enter your favorite fruit: ";
cin.getline(fruits, 50); // implementing getline() function
std::cout << "\nYour favorite fruit is :"<<fruits << std::endl;
return 0;
}
```

Output

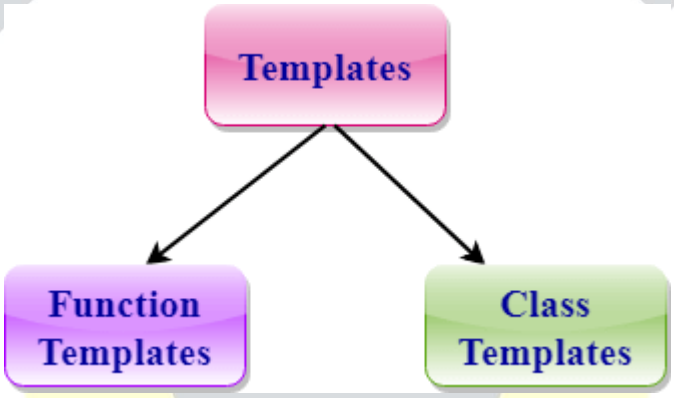
```
Enter your favorite fruit: Watermelon
Your favorite fruit is: Watermelon
```

C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.

- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
   // body of function.
}
```

Where,

Ttype: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
 #include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;

}
int main()
{
  int i =2;
  int j =3;
```

```
  float m = 2.3;
  float n = 1.2;
  cout<<"Addition of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<"Addition of m and n is :"<<add(m,n);
  return 0;
}
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
 template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
    // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
  fun(15,12.3);

  return 0;
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
#include <iostream>
```

```
using namespace std;
template<class X> void fun(X a)
{
   std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
   std::cout << "Value of b is : " <<b<< std::endl;
   std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
  fun(10);
  fun(20,30.5);
  return 0;
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

Class Template

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

where,

class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
   T num1 = 5;
   T num2 = 6;
```

```
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }

};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

Output:

```
Addition of num1 and num2 : 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

Class Template With Multiple Parameters

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, .....>
class class_name
{
  // Body of the class.
}
```

Let's see a simple example when class template contains two generic data types.

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
       T1 a;
       T2 b;
       public:
      A(T1 x,T2 y)
     {
        a = x;
        b = y;
     }
        void display()
       {
             std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
        }
    };

    int main()
   {
        A<int,float> d(5,6.5);
        d.display();
        return 0;
    }
```

Output:

Values of a and b are : 5,6.5

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.