PART 2 C++ Object Oriented

C++ Classes and Objects

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A **class** is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

C++ Class Definitions

When you define a class, you define an architecture for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows –

```
class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box –

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear –

```
#include <iostream>

using namespace std;

class Box {

   public:

      double length;   // Length of a box

      double breadth;  // Breadth of a box

      double height;   // Height of a box

};

int main() {

   Box Box1;        // Declare Box1 of type Box

   Box Box2;        // Declare Box2 of type Box

   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification

   Box1.height = 5.0;

   Box1.length = 6.0;

   Box1.breadth = 7.0;

   // box 2 specification

   Box2.height = 10.0;

   Box2.length = 12.0;

   Box2.breadth = 13.0;


   // volume of box 1

   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;


   // volume of box 2

   volume = Box2.height * Box2.length * Box2.breadth;

   cout << "Volume of Box2 : " << volume <<endl;

   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below −

Sr.No	Concept & Description
1	**Class Member Functions** A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
2	**Class Access Modifiers** A class member can be defined as public, private or protected. By default members would be assumed as private.
3	**Constructor & Destructor** A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

4	**Copy Constructor** The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
5	**Friend Functions** A **friend** function is permitted full access to private and protected members of a class.
6	**Inline Functions** With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.
7	**this Pointer** Every object has a special pointer **this** which points to the object itself.
8	**Pointer to C++ Classes** A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.
9	**Static Members of a Class** Both data members and function members of a class can be declared as static.

Constructors in C++

What is constructor?

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create.It is special member function of the class.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Types of Constructors

1. **Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
// Cpp program to illustrate the

// concept of Constructors

#include <iostream>

using namespace std;

class construct

{

public:

    int a, b;


        // Default Constructor

    construct()

    {

        a = 10;

        b = 20;

    }

};

int main()

{

  // Default constructor called automatically when the object is created

    construct c;

    cout << "a: "<< c.a << endl << "b: "<< c.b;

    return 1;

}
```

```
Output: a: 10
b: 20
```

Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
// CPP program to illustrate
// parameterized constructors
#include<iostream>
using namespace std;

class Point
{
    private:
        int x, y;
    public:
        // Parameterized Constructor
        Point(int x1, int y1)
        {
            x = x1;
            y = y1;
        }

        int getX()
        {
            return x;
        }
        int getY()
        {
            return y;
        }
    };

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

1. When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
2.   Example e = Example(0, 50); // Explicit call
3.   Example e(0, 50);           // Implicit call
```

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

Can we have more than one constructors in a class?
Yes, It is called Constructor Overloading.

4. **Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.

What is a copy constructor?

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Following is a simple example of copy constructor.

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15
```

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.

4. When compiler generates a temporary object.

Destructors in C++

What is destructor?

Destructor is a member function which destructs or deletes an object.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:
(1) the function ends
(2) the program ends
(3) a block containing local variables ends
(4) a delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)
Destructors don't take any argument and don't return anything

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};
String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}
String::~String()
{
    delete []s;
}
```

Data Abstraction in C++

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this –

```
#include <iostream>

using namespace std;



int main() {

   cout << "Hello C++" <<endl;

   return 0;

}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels –

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages –

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example –

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


class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }
      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Designing Strategy

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements −

- **Program statements (code)** − This is the part of a program that performs actions and they are called functions.
- **Program data** − The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are **private**. For example −

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Access Modifiers in C++

Access modifiers are used to implement important feature of Object Oriented Programming known as **Data Hiding**. Consider a real life example: What happens when a driver applies brakes? The car stops. The driver only knows that to stop the car, he needs to apply the brakes. He is unaware of how actually the car stops. That is how the engine stops working or the internal implementation on the engine side. This is what data hiding is.
Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.
Let us now look at each one these access modifiers in details:

- **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
// C++ program to demonstrate public
// access modifier

#include<iostream>
```

```
using namespace std;

// class definition
class Circle
{
    public:
        double radius;

        double compute_area()
        {
            return 3.14*radius*radius;
        }

};

// main function
int main()
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is:" << obj.radius << "\n";
    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

- Output:

```
▪ Radius is:5.5

▪ Area is:94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

- **Private**: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class. Example:

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double compute_area()
        {   // member function can access private
            // data member radius
            return 3.14*radius*radius;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class. **Output**:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
         double radius;
                ^
31:9: error: within this context
     obj.radius = 1.5;
         ^
```

However we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double compute_area(double r)
        {   // member function can access private
            // data member radius
            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is:" << radius << endl;
            cout << "Area is: " << area;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);


    return 0;
}
```

- **Output**:

```
Radius is:1.5

Area is: 7.065
```

- **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
    // protected data members
    protected:
    int id_protected;

};

// sub class or derived class
class Child : public Parent
{

    public:
    void setId(int id)
    {

        // Child class is able to access the inherited
        // protected data members of base class

        id_protected = id;

    }

    void displayId()
```

```
        {
            cout << "id_protected is:" << id_protected << endl;
        }
    };

    // main function
    int main()  {

        Child obj1;

        // member function of derived class can
        // access the protected data members of base class

        obj1.setId(81);
        obj1.displayId();
        return 0;
    }
```

Output:

```
id_protected is:81
```

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example –

```
#include <iostream>

using namespace std;


class Adder {

   public:

      // constructor

      Adder(int i = 0) {

         total = i;

      }


      // interface to outside world

      void addNum(int number) {

         total += number;

      }
      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Designing Strategy

Most of us have learnt to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This is applied most frequently to data members, but it applies equally to all members, including virtual functions.

Friend class and function in C++

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node

{

private:

  int key;

  Node *next;

  /* Other members of Node Class */

  friend class LinkedList; // Now class  LinkedList can

                             // access private members of Node

};
```

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:
a) A method of another class
b) A global function

```
class Node
{
private:
  int key;
  Node *next;

  /* Other members of Node Class */
  friend int LinkedList::search(); // Only search() of linkedList
                                   // can access internal members
};
```

Following are some important points about friend functions and classes:
1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
2) Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.
3) Friendship is not inherited
4) The concept of friends is not there in Java.

A simple and complete C++ program to demonstrate friend Class

```
#include <iostream>
```

```
class A {
private:
    int a;
public:
    A() { a=0; }
    friend class B;     // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x) {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};

int main() {
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

Output:

```
A::a=0
```

A simple and complete C++ program to demonstrate friend function of another class

```
#include <iostream>

class B;

class A
{
public:
    void showB(B& );
};

class B
{
private:
    int b;
public:
    B()  {  b = 0; }
    friend void A::showB(B& x); // Friend function
};

void A::showB(B &x)
{
    // Since show() is friend of B, it can
    // access private members of B
    std::cout << "B::b = " << x.b;
}
```

```
int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

C++ Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>


using namespace std;
```

```
// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

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

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes

Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private**members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private**members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax −

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

```
#include <iostream>


using namespace std;


// Base class Shape

class Shape {

   public:

      void setWidth(int w) {

         width = w;

      }

      void setHeight(int h) {

         height = h;

      }

   protected:

      int width;

      int height;

};


// Base class PaintCost

class PaintCost {

   public:

      int getCost(int area) {

         return area * 70;

      }

};


// Derived class
```

```
class Rectangle: public Shape, public PaintCost {

   public:

      int getArea() {

         return (width * height);

      }

};


int main(void) {

   Rectangle Rect;

   int area;


   Rect.setWidth(5);

   Rect.setHeight(7);


   area = Rect.getArea();


   // Print the area of the object.

   cout << "Total area: " << Rect.getArea() << endl;


   // Print the total cost of painting

   cout << "Total paint cost: $" << Rect.getCost(area) << endl;


   return 0;

}
```

When the above code is compiled and executed, it produces the following result –

```
Total area: 35
Total paint cost: $2450
```

C++ Overloading (Operator and Function)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types −

```
#include <iostream>

using namespace std;



class printData {

   public:

      void print(int i) {

        cout << "Printing int: " << i << endl;

      }

      void print(double  f) {

        cout << "Printing float: " << f << endl;

      }

      void print(char* c) {
         cout << "Printing character: " << c << endl;

       }

};


int main(void) {

   printData pd;


   // Call print to print integer

   pd.print(5);


   // Call print to print float

   pd.print(500.263);


   // Call print to print character

   pd.print("Hello C++");


   return 0;

}
```

When the above code is compiled and executed, it produces the following result –

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below −

```
#include <iostream>

using namespace std;


class Box {

   public:

      double getVolume(void) {

         return length * breadth * height;

      }

      void setLength( double len ) {

         length = len;

      }

      void setBreadth( double bre ) {

         breadth = bre;

      }

      void setHeight( double hei ) {

         height = hei;

      }


      // Overload + operator to add two Box objects.

      Box operator+(const Box& b) {
         Box box;

         box.length = this->length + b.length;

         box.breadth = this->breadth + b.breadth;

         box.height = this->height + b.height;

         return box;

      }


   private:

      double length;      // Length of a box

      double breadth;     // Breadth of a box

      double height;      // Height of a box

};


// Main function for the program

int main() {

   Box Box1;                // Declare Box1 of type Box

   Box Box2;                // Declare Box2 of type Box

   Box Box3;                // Declare Box3 of type Box

   double volume = 0.0;     // Store the volume of a box here


   // box 1 specification

   Box1.setLength(6.0);

   Box1.setBreadth(7.0);

   Box1.setHeight(5.0);


   // box 2 specification

   Box2.setLength(12.0);

   Box2.setBreadth(13.0);

   Box2.setHeight(10.0);
   // volume of box 1

   volume = Box1.getVolume();

   cout << "Volume of Box1 : " << volume <<endl;


   // volume of box 2

   volume = Box2.getVolume();

   cout << "Volume of Box2 : " << volume <<endl;


   // Add two object as follows:

   Box3 = Box1 + Box2;


   // volume of box 3

   volume = Box3.getVolume();

   cout << "Volume of Box3 : " << volume <<endl;


   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--

<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

Sr.No	Operators & Example
1	**Unary Operators Overloading**
2	**Binary Operators Overloading**
3	**Relational Operators Overloading**
4	**Input/Output Operators Overloading**
5	**++ and -- Operators Overloading**
6	**Assignment Operators Overloading**
7	**Function call () Operator Overloading**
8	**Subscripting [] Operator Overloading**

9	**Class Member Access Operator -> Overloading**

Polymorphism in C++

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes –

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();
   // store the address of Triangle

   shape = &tri;



   // call triangle area.

   shape->area();



   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Parent class area :
Parent class area :
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this −

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Interfaces in C++ (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows −

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

Abstract Class Example

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()** −

```
#include <iostream>



using namespace std;

```

```
// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

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


int main(void) {
   Rectangle Rect;
   Triangle  Tri;


   Rect.setWidth(5);
   Rect.setHeight(7);


   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;
   Tri.setWidth(5);
   Tri.setHeight(7);
   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Total Rectangle area: 35
Total Triangle area: 17
```

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

Designing Strategy

An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.

The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.

This architecture also allows new applications to be added to a system easily, even after the system has been defined.