OOP in C++

Copy Constructor in C++

```
// C++ program to demonstrate the working
// of a 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& p1)
    {
        x = p1.x;
        y = p1.y;
    }

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

OOP in C++

```
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
```

two types : explicit copy & implicit copy

```
// Example: Implicit copy constru

#include<iostream>
using namespace std;

class Sample
{
      int id;
```

```
    public:
    void init(int x)
    {
        id=x;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;
    obj2.display();
    return 0;
}
```

Output

```
ID=10

ID=10
```

```
// Example: Explicit copy constructor

#include<iostream>
```

OOP in C++

```
using namespace std;

class Sample
{
    int id;
    public:
    void init(int x)
    {
        id=x;
    }
    Sample(){}  //default constructor with empty body

    Sample(Sample &t)   //copy constructor
    {
        id=t.id;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};
int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;    copy constructor called
    obj2.display();
    return 0;
```

```
}
```

Output

```
ID=10

ID=10
```

Shallow Copy and Deep Copy in C++

	Shallow copy	deep copy
1.	When we create a copy of object by copying data of all member variables as it is, then it is called shallow copy	When we create an object by copying data of another object along with the values of memory resources that reside outside the object, then it is called a deep copy
2.	A shallow copy of an object copies all of the member field values.	Deep copy is performed by implementing our own copy constructor.
3.	In shallow copy, the two objects are not independent	It copies all fields, and makes copies of dynamically allocated memory pointed to by the fields
4.	It also creates a copy of the dynamically allocated objects	If we do not create the deep copy in a rightful way then the copy will point to

OOP in C++

the original, with disastrous consequences.

Shallow copy :

```
using namespace std;

// Box Class
class box {
private:
    int length;
    int breadth;
    int height;

public:
    // Function that sets the dimensions
    void set_dimensions(int length1, int breadth1,
                        int height1)
    {
        length = length1;
        breadth = breadth1;
        height = height1;
    }

    // Function to display the dimensions
    // of the Box object
    void show_data()
```

```
    {
        cout << " Length = " << length
             << "\n Breadth = " << breadth
             << "\n Height = " << height
             << endl;
    }
};

// Driver Code
int main()
{
    // Object of class Box
    box B1, B3;

    // Set dimensions of Box B1
    B1.set_dimensions(14, 12, 16);
    B1.show_data();

    // When copying the data of object
    // at the time of initialization
    // then copy is made through
    // COPY CONSTRUCTOR
    box B2 = B1;
    B2.show_data();

    // When copying the data of object
    // after initialization then the
    // copy is done through DEFAULT
    // ASSIGNMENT OPERATOR
    B3 = B1;
```

```
    B3.show_data();

    return 0;
}
```

Output:

```
Length = 14
Breadth = 12
Height = 16
Length = 14
Breadth = 12
Height = 16
Length = 14
Breadth = 12
Height = 16
```

Deep copy :

```
using namespace std;

// Box Class
class box {
private:
    int length;
```

```
    int* breadth;
    int height;

public:
    // Constructor
    box()
    {
        breadth = new int;
    }

    // Function to set the dimensions
    // of the Box
    void set_dimension(int len, int brea,
                       int heig)
    {
        length = len;
        *breadth = brea;
        height = heig;
    }

    // Function to show the dimensions
    // of the Box
    void show_data()
    {
        cout << " Length = " << length
             << "\n Breadth = " << *breadth
             << "\n Height = " << height
             << endl;
    }
```

```
    // Parameterized Constructors for
    // for implementing deep copy
    box(box& sample)
    {
        length = sample.length;
        breadth = new int;
        *breadth = *(sample.breadth);
        height = sample.height;
    }

    // Destructors
    ~box()
    {
        delete breadth;
    }
};

// Driver Code
int main()
{
    // Object of class first
    box first;

    // Set the dimensions
    first.set_dimension(12, 14, 16);

    // Display the dimensions
    first.show_data();

    // When the data will be copied then
```

```
    // all the resources will also get
    // allocated to the new object
    box second = first;

    // Display the dimensions
    second.show_data();

    return 0;
}
```

Output:

```
Length = 12
Breadth = 14
Height = 16
Length = 12
Breadth = 14
Height = 16
```

C++ Enumeration

In this article, you will learn to work with enumeration (enum). Also, you will learn where enums are commonly used in C++ programming.

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.

OOP in C++

Example 1: Enumeration Type

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

enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    return 0;
}
```

Output

```
Day 4
```

Example2: Changing Default Value of Enums

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

enum seasons { spring = 34, summer = 4, autumn = 9, winter = 32};

int main() {

    seasons s;

    s = summer;
    cout << "Summer = " << s << endl;

    return 0;
}
```

Output

```
Summer = 4
```

What is a C++ abstract class?

By definition, an **abstract class in C++** is a class that has at least *one* pure virtual function (i.e., a function that has no definition). The classes inheriting the abstract class *must* provide a definition for the pure virtual function; otherwise, the subclass would become an abstract class itself.

Abstract classes are essential to providing an abstraction to the code to make it reusable and extendable. For example, a *Vehicle* parent class with *Truck* and *Motorbike* inheriting from it is an abstraction that easily allows more vehicles to be added. However, even though all vehicles have wheels, not all vehicles have the same number of wheels – this is where a pure virtual function is needed.

```
#include <iostream>

using namespace std;

class Shape {
   public:
```

OOP in C++

```
      virtual int Area() = 0; // Pure virtual function is declared as
follows.
      // Function to set width.
      void setWidth(int w) {
         width = w;
      }
      // Function to set height.
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// A rectangle is a shape; it inherits shape.
class Rectangle: public Shape {
   public:
      // The implementation for Area is specific to a rectangle.
      int Area() {
         return (width * height);
      }
};
// A triangle is a shape too; it inherits shape.
class Triangle: public Shape {
   public:
      // Triangle uses the same Area function but implements it to
      // return the area of a triangle.
      int Area() {
         return (width * height)/2;
      }
};
```

OOP in C++

```
int main() {
  Rectangle R;
  Triangle T;

  R.setWidth(5);
  R.setHeight(10);

  T.setWidth(20);
  T.setHeight(8);

  cout << "The area of the rectangle is: " << R.Area() << endl;
  cout << "The area of the triangle is: " << T.Area() << endl;
}
```

Note: The return type of the virtual function must be consistent throughout all of its implementing classes.

You can't create an object of an abstract class type. However, you can use pointers and references to abstract class types.

Abstract class = interface

`const` values

The **`const`** keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it.

C++Copy

```
// constant_values1.cpp
int main() {
```

```
    const int i = 5;
    i = 10;   // C3892
    i++;   // C2105
}
```

Error

C++ Exceptions

C++ try and catch

Exception handling in C++ consist of three keywords: `try`, `throw` and `catch`:

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

```
try {
  int age = 15;
  if (age >= 18) {
```

```
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

Example

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Error number: " << myNum;
}
```

OOP in C++

9 C++ data structures you need to know