Object-Oriented Programming in C++

Note by Tawhid Monowar

Index **Page**

1. Class and Objects ———————————————————— 01
2. Methods ———————————————————— 02
3. Constructors ———————————————————— 02
4. Destructors ———————————————————— 03
5. Access Specifiers ———————————————————— 04
6. Inheritance ———————————————————— 04
7. Polymorphism ———————————————————— 05
8. Function Overloading ———————————————————— 07
9. Operator Overloading ———————————————————— 07
10. Encapsulation ———————————————————— 08
11. Abstraction ———————————————————— 09
12. Templates ———————————————————— 10
13. Namespaces ———————————————————— 11
14. Friend Class and Function ———————————————————— 12

1. Class and Objects

A class is a template for objects, and an object is an instance of a class. Example: Consider a class as a classroom, where each student inside the classroom is an object of that class.

Example: 1

```cpp
#include <bits/stdc++.h>
using namespace std;

class Classroom {
public:
      string name;
      vector<string> students;

      void displayStudents() {
      cout << "Classroom: " << name << endl;
      for (auto student : students) {
            cout << "Student: " << student << endl;
         }
      }

      void addStudent(string studentName) {
         students.push_back(studentName);
      }
};

int main() {

      Classroom classroom; // Create an object
      classroom.name = "Class A";
      classroom.addStudent("John");
      classroom.addStudent("Jane");
      classroom.addStudent("Alice");
      classroom.displayStudents();

      return 0;
}
```

Output: 1

```
Classroom: Class A
Student: John
Student: Jane
Student: Alice
```

2. Methods

Methods are functions that belongs to the class. Methods in a class are functions that are defined either inside or outside the class definition.

Example: 2

```
class PrintString {
  public:
      void print(string str) {
      cout << str;
    }
};

int main() {

  PrintString print_string; // Create an object
  print_string.print("Hello World!");

  return 0;
}
```

Output: 2

```
Hello World!
```

3. Constructor

A constructor is a special method that is automatically called when an object of a class is created. To create a constructor in C++ use the same name as the class. Constructors have no return type.

Example: 3

```
 class MyClass {   // The class
  public:
      MyClass () {  // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;   // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

Output: 3

```
Hello World!
```

4. Destructors

A destructor is a special method that is automatically called when an object is going to be destroyed. To create a destructor in C++ use the same name as the class preceded by a tilde (~) symbol. It is not possible to define more than one destructor. Destructor neither requires any argument nor returns any value. It is automatically called when an object goes out of scope. Destructor release memory space occupied by the objects created by the constructor. In a destructor, objects are destroyed in the reverse of an object creation.

Destructors with the access modifier as private are known as Private Destructors. Whenever we want to prevent the destruction of an object, we can make the destructor private.

Example: 4

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

class Test
{
public:
      Test()
      {
          cout<<"\n Constructor executed";
      }

      ~Test()
      {
          cout<<"\n Destructor executed";
      }
};

/*
private:
      ~Test() {}   // Private Destructor
};
*/

main()
{
      Test t;
      return 0;
}
```

Output: 4

```
 Constructor executed
 Destructor executed
```

5. Access Specifiers

Access specifiers in C++ determine the visibility of class members. There are three access specifiers: public, private, and protected. Public members are accessible from anywhere, private members are only accessible within the class, and protected members are accessible within the class and its derived classes.

Example: 5

```
class MyClass {
  Public:       // Public access specifier
      int x;    // Public attribute
  private:      // Private access specifier
       int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;   // Allowed (public)
  myObj.y = 50;   // Not allowed (private)
  return 0;
}
```

Output: 5

```
error: y is private
```

6. Inheritance

Inheritance is a way to create new classes based on existing classes. The new class, called the "derived class," inherits the properties of the existing class, known as the "base class." It can add its own features without changing the base class. The base class is like a parent, and the derived class is like a child that inherits traits from its parent.

Types of Inheritance in C++

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.
2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.
3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.
4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

Example: 6

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

// Base class
class Vehicle {
  public:
      string brand = "Ford";
      void honk() {
          cout << "Tuut, tuut! \n" ;
      }
};

// Derived class
class Car: public Vehicle {
  public:
       string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

Output: 6

```
Tuut, tuut!
Ford Mustang
```

7. Polymorphism

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism.

Types of Polymorphism

1. Compile-time Polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.
2. Runtime Polymorphism: This type of polymorphism is achieved by Function Overriding. Late binding and dynamic polymorphism are other names for runtime polymorphism.

Example: 7

```
#include <bits/stdc++.h>
using namespace std;

class myClass{
public:
    // Function with 1 int parameter
    void func(int x)
    {
       cout << "value of x is " << x << endl;
    }

    // Function with same name but 1 double parameter
    void func(double x)
    {
       cout << "value of x is " << x << endl;
    }

    // Function with same name and 2 int parameters
    void func(int x, int y)
    {
       cout << "value of x and y is " << x << ", " << y << endl;
    }
};

int main()
{
    myClass obj1;
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);

    return 0;
}
```

Output: 7

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

8. Function Overloading

Function overloading is a feature of object-oriented programming (Polymorphism) where two or more functions can have the same name but different parameters.

Example:

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

void add(int a, int b)
{
    cout << "sum = " << (a + b);
}

void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}

int main()
{
    add(10, 2);
    add(5.3, 6.2);

    return 0;
}
```

Output: 8

```
sum = 12
sum = 11.5
```

9. Operator Overloading

Operator overloading in C++ enables us to redefine the behavior of operators for user-defined types. It allows us to use operators like +, -, *, etc., with objects of our custom classes, giving them special meanings.

For example, we can overload the + operator to concatenate strings, the - operator to subtract complex numbers, or the * operator to multiply matrices. This provides a more natural and intuitive way to work with objects of user-defined classes.

By overloading operators, we can leverage the power of familiar operators to perform operations specific to our custom classes, enhancing code readability and usability.

10. Encapsulation

Encapsulation is about hiding sensitive data from users. To do this, we declare class variables as private, which means they can't be accessed from outside the class. To allow access to these private variables, we create public "get" and "set" methods that let users read or modify the values in a controlled way.

Example: 10

```cpp
class Person {
private:
      string name;
      int age;

public:
      // Setter method for name and age
      void set(string newName, int newAge) {
         name = newName;
         age = newAge;
      }

      // Getter method for name
      string getName() const {
         return name;
      }

      // Getter method for age
      int getAge() const {
         return age;
      }
};

int main() {

      Person person;
      person.set("John Doe",25);

      cout << "Name: " << person.getName() << endl;
      cout << "Age: " << person.getAge() << endl;

      return 0;
}
```

Output: 10

```
Name: John Doe
Age: 25
```

11. Abstraction

Abstraction means displaying only essential information and hiding the details. The idea behind abstraction is to create classes that provide a high-level view of an object's functionality without exposing its internal workings.

Example: 11

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

// Abstract base class
class Shape {
public:
      virtual void draw() = 0;
};

// Concrete derived class
class Circle : public Shape {
public:
      void draw() {
          cout << "Drawing a circle." << endl;
      }
};

int main() {
      Circle circle;
      circle.draw(); // Outputs "Drawing a circle."
      return 0;
}
```

This code shows how abstraction works in C++. It defines a basic shape concept called Shape using an abstract base class. The Shape class has a function called draw() that doesn't have an implementation.

The code also introduces a specific shape called Circle that inherits from Shape and provides its own implementation of the draw() function.

In the main() function, a Circle object is created, and its draw() function is called, resulting in the output "Drawing a circle."

Abstraction allows us to create a blueprint for shapes and define common behaviors without worrying about the exact implementation. We can then derive specific shapes from the base class and provide their own implementations of those behaviors.

12. Template

A template is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types.

Example: 12.0

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

template <typename Temp>
Temp sum(Temp a, Temp b) {
      return a + b;
}

int main() {

      int intSum = sum(5, 10);
      double doubleSum = sum(3.14, 2.718);

      cout << "Sum of 5 and 10: " << intSum << endl;
      cout << "Sum of 3.14 and 2.718: " << doubleSum << endl;

      return 0;
}
```

Output: 12.0

```
Sum of 5 and 10: 15
Sum of 3.14 and 2.718: 5.858
```

Example: 12.1

```cpp
#include <iostream>

using namespace std;

template<class T>
void print(T value) {
  cout << value << endl;
}

int main() {
  print(10);   // output: 10
  print(12.34);   // output: 12.34
  print("Hello, world!");  // output: Hello, world!
  return 0;
}
```

13. Namespaces

A namespace is a feature that allows you to organize code elements such as variables, functions, and classes into separate named scopes. Namespaces help avoid naming conflicts between different components of a program.

Example: 13

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

// first name space
namespace first_space
{
  void func()
  {
      cout << "Inside first_space" << endl;
  }
}

// second name space
namespace second_space
{
  void func()
  {
      cout << "Inside second_space" << endl;
  }
}

using namespace first_space;

int main ()
{
   // This calls the function from the first name space.
  func();
  return 0;
}
```

Output: 13

```
Inside first_space
```

14: Friend Class and Function

A friend function is a function that is declared outside a class but can access the private and protected members of the class. Friend functions are declared using the `friend` keyword in the class declaration.

For example, the following code declares a friend function called `print_private_data()` :

Example: 14.0

```
class MyClass {
  private:
      int m_data;

  public:
      friend void print_private_data(MyClass& obj);
};

void print_private_data(MyClass& obj) {
    cout << obj.m_data << endl;
}
```

The `print_private_data()` function can access the private member `m_data` because it is declared as a friend function of the `MyClass` class.

Friend classes are similar to friend functions, but instead of declaring a friend function, you declare a friend class. This means that all the member functions of the friend class have access to the private and protected members of the original class.

For example, the following code declares a friend class called `MyFriendClass` :

Example: 14.1

```
class MyClass {
  private:
      int m_data;

  public:
      friend class MyFriendClass;
};

class MyFriendClass {
  public:
      void print_private_data(MyClass& obj) {
          cout << obj.m_data << endl;
      }
};
```

The `MyFriendClass` class can access the private member `m_data` because it is declared as a friend class of the `MyClass` class.