THEORY

Table of Contents

1. Class **2**
- 1.1. Access Modifiers 3
- 1.2. Encapsulation 4
- 1.3. Class vs Structure vs Union 5

2. Constructor **7**

3. Destructor **8**
- 3.1. Constructor vs Destructor 9

4. Object **10**
- 4.1. Array of Objects 11
- 4.2. Passing and Returning Objects 13

5. Pointer and Reference **16**

6. Polymorphism **17**

7. Inline Function **18**

8. Friend Function **19**

9. References **21**

1. Class

Classes and objects are the basic building block that leads to Object-Oriented programming in C++. A `class` is a **user-defined data type**, which holds its own **data members** and **member functions**, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

- An `Object` is an instance of a `Class`.
- A class is defined in C++ using the keyword `class` followed by the name of the class.

Syntax of a `Class`:

```
class ClassName
{
    access_specifier:
    // body of the class
};
```

Note

When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

1.1. Access Modifiers

Ans.: Access modifiers (or access specifiers) are **keywords** in object-oriented programming languages that set the accessibility of **classes**, **methods**, and **other members**. Access modifiers are a specific part of programming language syntax used to facilitate the **encapsulation** of components.

In C++, there are **three** access modifiers:

1. **Public**: All the class members declared under the **public** access modifier will be available to everyone.

2. **Private**: The class members declared as **private** can be accessed only by the member functions inside the class. Only the member functions or the **friend functions/classes** are allowed to access the private data members of the class.

3. **Protected:** The **protected** access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a **friend class**. The difference is that the class members declared as **protected** can be accessed by any subclass (derived class) of that class as well.

1.2. Encapsulation

Ans.: Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. For example,

```cpp
class Rectangle {
  public:
    int length;
    int breadth;

    int getArea() {
      return length * breadth;
    }
};
```

1.3. Class *vs* Structure *vs* Union

Ans. Here is comparison between **Class**, **Structure**, and **Union** in C++:

	Class	Structure	Union
Keyword	It is declared using the **class** keyword.	It is declared using the **struct** keyword.	It is declared using the **union** keyword.
Use Case	It is normally used for data abstraction and inheritance.	It is normally used for the grouping of different datatypes.	It is used to achieve memory efficiency when the available memory is limited.
Accessing Members	Members of a class are **private** by default.	Members of a structure are **public** by default.	Member variables of a union are **public** by default.
Size	Theoritically, the size of a class can be up to 2 to the power of 55. (2^55)	The size of a structure is the sum of the size of its data members.	The size of a union is equal to the size of the largest data type.

	Class	Structure	Union
Memory Allocation	It can store multiple values of the various members.	Same as Class.	It stores one value at a time for all of its members.
Value Altering	Altering the values of a single member does not affect the other members of a Class.	Altering the values of a single member does not affect the other members of a Structure.	Altering the values of a single member, it affects the values of other members.
Initialization	We can initialize multiple members at the same time.	Same as Class.	We can only initiate the first member at a time.

2. Constructor

`Constructor` in C++ is a special method that is invoked automatically **at the time an object of a class is created**. It is used to initialize the data members of new objects generally.

The prototype of the constructor looks like this:

```cpp
class ClassName()
{
    // ...
public:
    ClassName()
    {
        // ...
    }
}
```

Types of Constructor:

- **Default Constructor:** A default constructor is a constructor that doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.
- **Parameterized Constructor:** Parameterized constructors make it possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.
- **Copy Constructor:** A copy constructor is a **member function** that initializes an object using another object of the same class. Copy constructor takes a reference to an object of the same class as an argument.

3. Destructor

A destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is **the last function** that is going to be called before an object is destroyed.

A destructor has the same name as the class and is preceded by a **tilde (~)**. For example, the destructor for class `ClassName` is declared as `~ClassName()`

```
class ClassName()
{
    // ...
public:
    ~ClassName()
    {
        // destructor code here...
    }
}
```

Characteristics of a destructor:

- Destructor neither requires any argument nor returns any value.
- It is not possible to define more than one destructor. Simply put, **destructor cannot be overloaded**.
- It cannot be declared static or const.

3.1. Constructor vs Destructor

Here is a comparison table between **Constructor** and **Destructor**:

Constructor	Destructor
Constructor is a member function that has the same name as the name of the class.	Destructors are typically used to deallocate memory.
When the object is created, a constructor is called automatically.	When the program gets terminated, the destructor is called automatically.
A constructor allows an object to initialize some of its value before it is used.	A destructor allows an object to execute some code at the time of its destruction.
There can be various constructors in a class	There is constantly a single destructor in the class
Can be overloaded.	Cannot be overloaded.
Receives arguments.	Does not receive any argument.

4. Object

An Object is an instance of a Class.

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, we need to create objects.

4.1. Array of Objects

The Array of Objects stores `objects`. An array of a class type is also known as an array of objects.

```
ClassName ObjectName[number of objects];
```

We can store information of multiple employees using an array of objects. For example,

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

class Person
{
    string name;
    string position;

public:
    Person(string n, string p)
    {
        name = n;
        position = p;
    }

    string getName()
    {
        return name;
    }
};

int main()
{
    Person team[10] = {
        Person("Shayan", "TR"),
        Person("Munna", "E"),
        Person("Simon", "E"),
        Person("Sharmin", "E"),
        Person("Fatema", "E"),
        Person("Ahona", "E"),
        Person("Rubaiyat", "E"),
        Person("Rebeka", "E"),
        Person("Bulbul", "E"),
        Person("Jamal", "HR")};

    for (int i = 0; i < 10; i++)
        cout << team[i].getName() << endl;

    return 0;
}
```

Output: The above code yields the following output in the terminal:

```
Shayan
Munna
Simon
Sharmin
Fatema
Ahona
Rubaiyat
Rebeka
Bulbul
Jamal
```

4.2. Passing and Returning Objects

Below is an example of passing an object:

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

class Person
{
public:
    string name;
    int age;
};

void displayPersonInformation(Person person)
{
    cout << person.name << endl;
    cout << person.age << endl;
}

int main()
{
    Person p1;
    p1.name = "Shahriar";
    p1.age = 21;

    displayPersonInformation(p1);
    return 0;
}
```

Also, here is how we return an object:

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

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

public:
    Person(string n, int a)
    {
        name = n;
        age = a;
    }

    void introduce()
    {
        cout << "Hey, it's " << name << ", and I am " <<
age << " years old." << endl;
    }
};

Person returnObject(string name, int age)
{
    Person person(name, age);
    return person;
}

int main()
{
    Person p = returnObject("Shahriar", 21);
    p.introduce();
    return 0;
}
```

We can also use a **Copy Constructor** to return objects:

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

```
class Point
{
private:
    int x, y;

public:
    Point(int a, int b)
    {
        x = a;
        y = b;
    }

    Point(const Point &p)
    {
        x = p.x;
        y = p.y;
        cout << "Copy constructor called!" << endl;
    }

    void display()
    {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main()
{
    Point p1(10, 20);
    cout << "Point p1: ";
    p1.display();

    Point p2 = p1;
    cout << "Point p2: ";
    p2.display();

    return 0;
}
```

5. Pointer And Reference

Object Pointer: Pointer to Object in C++ is defined as the pointer that is used for accessing objects.

Reference Object: A reference, like a pointer, stores the address of an object that is located elsewhere in memory. Unlike a pointer, a reference after it's initialized can't be made to refer to a different object or set to null.

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

class Person
{
public:
    string name;
    void getName()
    {
        cout << name << endl;
    }
};

int main()
{
    Person person;
    Person *ptr = &person;

    ptr->name = "Shahriar";
    ptr->getName();
    return 0;
}
```

6. 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. In C++, function overloading is a great example of polymorphism.

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

```cpp
int add(int a, int b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

double add(double a, double b, double c) {
    return a + b + c;
}
```

7. Inline Function

Inlining is a manual or compiler optimization that replaces a function call site with the body of the called function.

An **inline function** is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. A function can be inlined using the `inline` keyword before a function:

```
inline int add(int a, int b, int c) {
    return a + b + c;
}
```

We know that all the functions defined inside a `class` are automatically inlined. Here is an example code of automatic inlining using a `class` :

```
class Samp
{
public:
    int getData(int id)
    {
        // this function is automatically inline
        // function body
    }
};
```

8. Friend Function

In object-oriented programming, a friend function, that is a *friend* of a given class, is a function that is given the same access as methods to private and protected data.

A friend function can access the private and protected data of a class. We declare a friend function using the `friend` keyword inside the body of the class.

```cpp
class className
{
    friend returnType functionName(parameters);
}
```

Here is a C++ code that demonstrates the functionality of a friend function:

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

class Distance
{
private:
    int meter;
    friend int addFive(Distance);

public:
    Distance() : meter(0) {}
};

// friend function definition
int addFive(Distance d)
{
    // accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main()
{
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output: The above code yields the following output in the terminal:

```
Distance: 5
```

9. References

- **Wikipedia:** Access modifiers
- **GeeksforGeeks:** Access Modifiers in C++
- **Programiz:** Encapsulation in C++ (with Examples)
- **GeeksforGeeks:** Difference Between Structure and Class in C++
- **Byjus**: Difference Between Structure and Union in C
- **Itanium C++ ABI draft**: Section 1.2
- **GeeksforGeeks**: Destructors in C++
- **Byjus:** Difference Between Constructor and Destructor in C++
- **GeeksforGeeks**: Array of Objects in C++ with Examples
- **GeeksforGeeks**: Polymorphism in C++
- **GeeksforGeeks**: Inline Functions in C++
- **Programiz:** C++ Friend Functions and Classes (with Examples)
- **GeeksforGeeks:** References in C++