const and static in C++

const in C++

- The const keyword is used to define **constant values, functions, and objects** that cannot be modified after initialization.

const Variable

- A const variable is a **read-only** variable whose value cannot be changed after initialization.

Example: Declaring and Using const Variables

```
int main() {
   const int x = 10;  // Constant integer
   cout << "x = " << x << endl;

   // x = 20;  // Error: assignment of read-only variable 'x'

   return 0;
}
```

- const variables **must** be initialized when declared.
- They **cannot be modified** later.
- Used to prevent accidental modifications.

const Function

A const function in a class **does not modify the object's state**. It is mainly used in **getter functions**.

Example: const Function in a Class

```
class Example {
private:
    int x;

public:
    Example(int val) { x = val; }

    void setX(int val) { x = val; }

    int getX() const {  // `const` function
        return x;
    }
};
```

- const functions **cannot modify any member variables**.
- They are useful for getter methods.
- If you try modifying any member inside a const function, **a compiler error occurs**.

const Object

A const object is an instance of a class **that cannot modify its own data**.

Example: Declaring a const Object

```
class Example {
private:
   int x;

public:
   Example(int val) { x = val; }

   int getX() const { return x; } // `const` function

   void setX(int val) { x = val; } // Non-const function
};

int main() {
   const Example obj(10); // `const` object

   cout << obj.getX() << endl; // Allowed (const function)

   // obj.setX(20); // Error: Cannot call a non-const function on a const object

   return 0;
}
```

- A const object can **only** call const functions.
- It **cannot** modify its own members.
- Useful for **ensuring immutability** in a program.

static in C++

The static keyword defines **class-wide** members that belong to the class itself rather than any particular instance.

static Variable

A static variable inside a class is **shared among all instances** of that class.

Example: Using a static Variable

```
class Example {
private:
   static int count;  // Static variable

public:
   Example() { count++; }

   static int getCount() { return count; }  // Static function
};

// Definition of static variable (outside class)
int Example::count = 0;

int main() {
   Example obj1, obj2, obj3;
   cout << "Total Objects: " << Example::getCount() << endl;  // Output: 3

   return 0;
}
```

- static variables are **shared** across all instances of the class.
- They are declared **inside the class** but **defined outside** the class.
- They **retain their value** between function calls.

static Function

A static function **belongs to the class** rather than any instance. It can only access **static members**.

Example: Declaring and Using a static Function

```
class Example {
private:
   static int count;

public:
   static void showCount() {
      cout << "Count: " << count << endl;
   }
};

// Definition of static variable
int Example::count = 10;

int main() {
   Example::showCount();  // Calling a static function

   return 0;
}
```

- static functions **do not require an object** to be called.
- They **cannot** access non-static members of the class.

Static Objects in C++

A **static object** is an object that persists throughout the program's execution. It is initialized **only once** and retains its value between function calls.

- **Lifetime:** A static object is created **only once** and destroyed at the end of the program.
- **Scope:** If declared inside a function, it remains **local to the function** but retains its value between calls.
- **Initialization:** It is initialized **only once** when encountered for the first time.

Example 1: Static Object Inside a Function

```
void test() {
    static Example obj;  // Static object (created once)
    obj.display();
}
```

```
class Example {
public:
    Example() {
        cout << "Constructor called" << endl;
    }
    ~Example() {
        cout << "Destructor called" << endl;
    }

    void display() {
        cout << "Hello from Example class!" << endl;
    }
};

void test() {
    static Example obj;  // Static object (created once)
    obj.display();
}

int main() {
    test();  // First call: Object is created and used
    test();  // Second call: Same object is reused (no new constructor call)
    return 0;
}
```

- The static object obj inside test() is **created only once**.
- Even though test() is called multiple times, the object is **not re-created**.
- The **destructor is called only when the program exits**.

Constructor called
Hello from Example class!
Hello from Example class!
Destructor called (after program exits)

Member Initialization List in C++

A **Member Initialization List** in C++ is used to initialize class data members before the constructor body executes. This is particularly useful for **const members, reference members, and base class initialization**.

```
class ClassName {

   int a;
   const int b;

public:
   // Constructor using Member Initialization List

   ClassName(int x, int y) : a(x), b(y) {
      cout << "Constructor called\n";
   }
};
```

Why Use Member Initialization List?

- **Efficient Initialization:** Avoids an extra assignment.
- **Mandatory for const and reference members:** These cannot be assigned later.
- **Base Class Initialization:** Ensures correct construction order.

```
class A {
   int x;
   const int y;  // Constant member
   int &z;      // Reference member

public:
   // Constructor with Member Initialization List
   A(int a, int b, int &c) : x(a), y(b), z(c) {
      cout << "x: " << x << ", y: " << y << ", z: " << z << endl;
   }
};

int main() {
   int num = 30;
   A obj(10, 20, num);
   return 0;
}
```

```
class Example {
   const int a;  // Constant member
   int &ref;     // Reference member

public:
   int value = 50;  // Normal member variable

   // Member Initialization List with Default Values
   Example() : a(100), ref(value) {
      cout << "a = " << a << ", ref = " << ref << endl;
   }
};

int main() {
   Example obj;  // Default constructor is called
   return 0;
}
```

•Initializing const and reference members.
•Optimized initialization for performance.
•Ensuring base classes are correctly initialized before derived class members.