OOPC-02-Function & Constructor

Function

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.
- Predefined Function: Library functions such as main()
- User-Defined Functions: Defined by users

Syntax

```
void myFunction()
{
    // code to be executed
}
```

Calling a Function

- Declared functions are not executed immediately, they will be executed later, when they are called
- To call a function, write the function's name followed by two parentheses () and a semicolon
- A function can be called multiple times

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

void myFunction()
{
    cout<<"HELLO WORLD!"<<endl;
}

int main()
{
    myFunction();
    myFunction();
    myFunction();
    return 0;
}
```

```
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
```

Function Declaration & Definition

- A function consists of two parts:
  - **Declaration**: the return type, the name of the function, and parameters (if any)
  - **Definition**: the body of the function (code to be executed)

> ⚠ **Caution**
>
> If a user-defined function, such as myFunction() is declared after the main() function, an error will occur:

Instead you can have function declaration above main(), and function definition below main().

```
// Function declaration
void myFunction();

int main()
{
    myFunction();
    return 0;
}

// Function definition
void myFunction()
{
    cout << "HELLO WORLD!";
}
```

```
HELLO WORLD!
```

Parameters & Arguments

- Information can be passed to functions as a parameter.
- Parameters act asvariables inside the function
- Parameters are specified after the function name, inside the parentheses.
- You can add as many parameters as you want

```
void myFunction(string fname)
{
    cout << "Hello! " << fname << endl;
}
```

- Here fname is Parameter
- When a parameter is passed to the function, it is called an argument.

```
int main()
{
```

```
    myFunction("Dipesh");
    myFunction("Nilesh");
    return 0;
}
```

```
Hello! Dipesh
Hello Nilesh
```

- Here, Dipesh and Nilesh is Arguments

Default Parameters

- You can also use a default parameter value, by using the equals sign (=)
- If we call the function without an argument, it will use the default value

```
void myFunction(string fname="John")
{
    cout << "Hello! " << fname << endl;
}
int main()
{
    myFunction();
    return 0;
}
```

```
Hello! John
```

> **Note**
>
> A parameter with a default value, is often known as an "optional parameter".

Multiple Parameters

- When you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

```
void myFunction(string fname, int age)
{
    cout << "Hello! " << fname << ". You are " << age << endl;
}
int main()
{
    myFunction("Dipesh", 19);
    myFunction("Nilesh", 20);
    return 0;
}
```

```
Hello! Dipesh. You are 19
Hello! Nilesh. You are 20
```

Call by Value

- In call by value, original value is not modified.
- The function receives a copy of the actual parameter values passed to it
- Any changes made to the parameters inside the function do not affect the original variables in the calling function
- The function works with its own local copies of the variables

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

// Function that swaps two integers
void swapByValue(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5, b = 10;
    cout << "Before swap: a = " << a << ", b = " << b << endl;

    // Call the swapByValue function
    swapByValue(a, b);

    cout << "After swap: a = " << a << ", b = " << b << endl;

    return 0;
}
```

```
Before swap: a = 5, b = 10
After swap: a = 5, b = 10
```

Call by Reference

- In this method function receives a reference to the actual variable rather than a copy of its value
- function can directly modify the original variable's value
- Any changes made inside the function are reflected in the variable in the calling function.

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

// Function that swaps two integers using call by reference
void swapByReference(int &x, int &y) {
    int temp = x;
```

```
    x = y;
    y = temp;
}

int main() {
    int a = 5, b = 10;
    cout << "Before swap: a = " << a << ", b = " << b << endl;

    // Call the swapByReference function
    swapByReference(a, b);

    cout << "After swap: a = " << a << ", b = " << b << endl;

    return 0;
}
```

```
Before swap: a = 5, b = 10
After swap: a = 10, b = 5
```

Inline 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.
- An inline function may increase efficiency if it is small.
  Syntax

```
inline return-type function-name(parameters)
{
    // function code
}
```

> **Remember**
>
> Inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

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

inline int cube(int s) { return s * s * s; }

int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

Static Data Member

- The static keyword is used with a variable to make the memory of the variable static once a static variable is declared its memory can't be changed
- Static members of a class are not associated with the objects of the class
- Once a static member is declared it will be treated as same for all the objects associated with the class

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

class Student
{
    public:
    // static member
    static int total;
    // Constructor called
    Student(){
        total += 1;
    }
};

int Student::total = 0;

int main()
{
    // Student 1 declared
    Student s1;
    cout << "Number of students:" << s1.total << endl;

    // Student 2 declared
    Student s2;
    cout << "Number of students:" << s2.total << endl;

    // Student 3 declared
    Student s3; cout << "Number of students:" << s3.total << endl;
    return 0;
}
```

Static Member Function

- Static Member Function in a class is the function that is declared as static because of which function attains certain properties:
    1. A static member function can be called even if no objects of the class exist
    2. A static member function can also be accessed using the class name through the scope resolution operator.
    3. A static member function can access static data members and static member functions inside or outside of the class
    4. Static member functions have a scope inside the class and cannot access the current object pointer

5. You can also use a static member function to determine how many objects of the class have been created

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

class Box {
    private:
        static int length;
        static int breadth;
        static int height;
    public:
        static void print(){
            cout << "The value of the length is: " << length << endl;
            cout << "The value of the breadth is: " << breadth << endl;
            cout << "The value of the height is: " << height << endl;
        }
};

// initialize the static data members
int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;

int main(){

    Box b;
    cout << "Static member function is called through Object name: \n" << endl;

    b.print();
    cout << "\nStatic member function is called through Class name: \n" << endl; Bo›

    return 0;
}
```

```
Static member function is called through Object name:
    The value of the length is: 10
    The value of the breadth is: 20
    The value of the height is: 30

Static member function is called through Class name:
    The value of the length is: 10
    The value of the breadth is: 20
    The value of the height is: 30
```

Why we need Static Member Function

- Static members are frequently used to store information that is shared by all objects in a class.
- For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be

increased each time an object is generated to keep track of the overall number of objects.

Friend Function

- A Friend Function is a function that can access and use the protected and private data of a class
- By using the keyword friend compiler knows the given function is a friend function
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend
  Syntax

```
class class_name
{
    friend data_type function_name(argument/s);
};
```

Characteristics of Friend Function

- The function is not in the scope of the class to which it has been declared as a friend
- It cannot be called using the object as it is not in the scope of that class
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

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

class Box
{
    private:
        int length;
    public:
        Box(): length(0) { }
        friend int printLength(Box); //friend function
};

int printLength(Box b)
{
    b.length += 10;
    return b.length;
}

int main()
{
    Box b;
    cout<<"Length of box: "<<printLength(b)<<endl;
    return 0;
}
```

```
OUTPUT: Length of Box: 10
```

Constructor

- A constructor in C++ is a special method that is automatically called when an object of a class is created
- To create a constructor, use the same name as the class, followed by parentheses ()
- The constructor has the same name as the class, it is always public, and it does not have any return value.

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

class MyClass {
public:
    MyClass() {
        cout << "HELLO WORLD!" << endl;
    }
};

int main() {
    MyClass myObj;
    return 0;
}
```

Default Constructor

- A constructor without any arguments or with the default value for every argument is said to be the Default constructor
- A constructor that accept no arguments is called a zero argument constructor or default constructor.
- If default constructor is not defined in the source code by the programmer, then the compiler defined the default constructor implicitly during compilation.
- They are used to create objects, which do not have any specific initial value.

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

class A{
    public:
    // User defined constructor
    A(){ cout << "A Constructor" << endl; }

    // uninitialized
    int size;
};
```

```
class B : public A{
    // compiler defines default constructor of B, and
    //inserts stub to call A constructor
    // compiler won't initialize any data of A
};

class C : public A {
    public: C(){
        // User defined default constructor of C
        // Compiler inserts stub to call A's constructor
        cout << "C Constructor" << endl;
        // compiler won't initialize any data of A
    }
};

class D {
    public: D() {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;
        // compiler won't initialize any data of 'a'
    }
    private:
    A a;
};

int main() {
    Base base;
    B b;
    C c;
    D d;
    return 0;
}
```

Parameterized Constructor

- A parameterized constructor in C++ is a constructor that accepts parameters during object initialization.
- It allows you to initialize the object's data members with specific values at the time of object creation.
- This is particularly useful when you want to create objects with different initial states.
- There is a minute difference between default constructor and Parametrized constructor.
    - The default constructor is a type of constructor which has no arguments but yes object instantiation is performed there also.
    - Parametrized constructor is a special type of constructor where an object is created, and further parameters are passed to distinct objects.

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

```cpp
class Rectangle {
private:
    int length;
    int width;

public:
    // Parameterized constructor
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }

    // Function to calculate area
    int area() {
        return length * width;
    }
};

int main() {
    // Create Rectangle object with width 5 and length 10
    Rectangle rect1(10, 5);

    // Calculate and print area of rect1
    cout << "Area of rectangle 1: " << rect1.area() << endl;

    // Create Rectangle object with width 7 and length 3
    Rectangle rect2(3, 7);

    // Calculate and print area of rect2
    cout << "Area of rectangle 2: " << rect2.area() << endl;

    return 0;
}
```

Copy Constructor

- A copy constructor in C++ is a special constructor used to create a new object as a copy of an existing object of the same class.
- It is called when an object is initialized with another object of the same class, either by direct initialization, passing by value, or returning by value.
- In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor
- Copy constructor takes a reference to an object of the same class as an argument.
- The process of initializing members of an object through a copy constructor is known as copy initialization.
- It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.

Characteristics of Copy Constrictor

- The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
- The process of initializing members of an object through a copy constructor is known as copy initialization
- It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis

Types of Copy Constructor

1. Default Copy Constructor
2. User Defined Copy Constructor

Default Copy Constructor

- The default copy constructor performs a member-wise copy of each data member from the source object to the destination object.
- An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

```
#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();

    // Implicit Copy Constructor Calling
    Sample obj2(obj1);
    obj2.display();
    return 0;
}
```

```
ID = 10
ID = 10
```

User-Defined Copy Constructor

- A user-defined copy constructor is generally needed when an object owns pointers or non-shareable references, such as to a file, in which case a destructor and an assignment operator should also be written

```cpp
#include <iostream>
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);
    obj2.display();
    return 0;
}
```

```
ID=10
ID=10
```

When is the Copy Constructor is called?

- In C++, a Copy Constructor may be called in the following cases:
    - When an object of the class is returned by value.
    - When an object of the class is passed (to a function) by value as an argument.
    - When an object is constructed based on another object of the same class.

Multiple Constructor

- Overloaded constructors essentially have the same name (exact name of the class) and different by number and type of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

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

class construct{
    public:
    float area;
    // Constructor with no parameters
```

```
    construct(){
    area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b){
        area = a * b;
    }

    void disp(){
        cout<< area<< endl;
    }
};

int main(){

    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

```
0
200
```

Destructors

- 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.
- Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor.
- Hence destructor can-not be overloaded.
- 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 destructor, objects are destroyed in the reverse of an object creation.

Syntax

```
//The syntax for defining the destructor within the class:
~ <class-name>() {
```

```
    // some instructions
}
```

```
//The syntax for defining the destructor outside the class:
<class-name> :: ~<class-name>() {
    // some instructions
}
```

Example:

```
#include <iostream>

using namespace std;
class Test {
  public:
    // User-Defined Constructor
    Test() {
      cout << "\n Constructor executed";
    }
    // User-Defined Destructor
    ~Test() {
      cout << "\nDestructor executed";
    }
};
main() {
  Test t;
  return 0;
}
```

```
Constructor executed
Destructor executed
```

Properties of Destructor

- The destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of the destructor.

When is the Destructor is called?

- A destructor function is called automatically when the object goes out of scope:
  - The function ends
  - The program ends
  - A block containing local variables ends

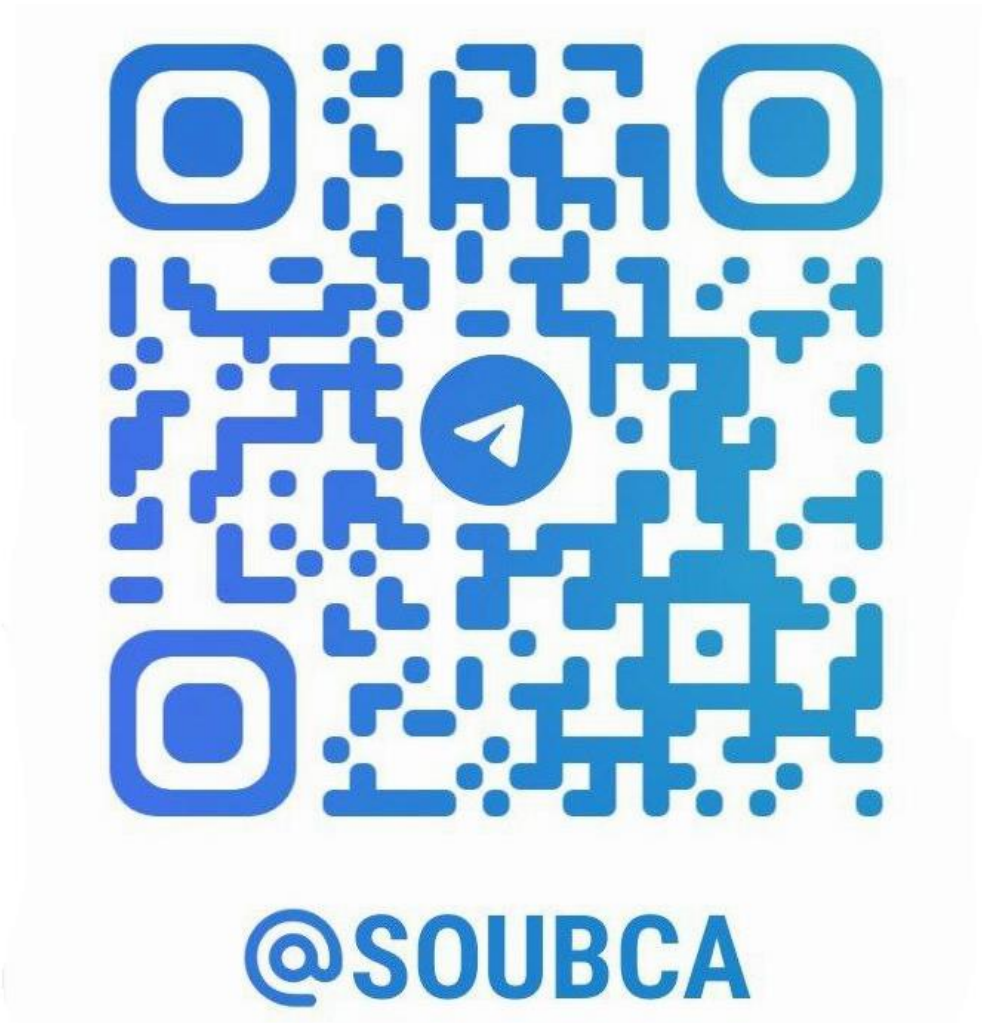
- A deleted operator is called

Call Destructor Explicitly

```
object_name.~class_name()
```

Questions

1. What is an Inline function? Explain with example. (MID-03M)
2. Compare and contrast call by value and call by reference. (MID-04M)
3. Write the characteristics of static member function.
4. Write the characteristics of static data members.
5. Explain friend function with an example. (MID-04M)
6. Differentiate between constructor and destructor.
7. Explain the use of constructor and its characteristics.
8. Define Default Constructor, Parameterized Constructor and Copy Constructor with examples.
9. Explain the concept of constructor overloading
10. How many types of constructors are supported in C++? Explain them with example (MID-05M)
11. Describe the purpose of destructor in C++. (MID-03M)
12. Explain the use of Static member Function in C++ (MID-03M)



Join Us on Telegram @SOU BCA