IEEE SCT SB

IEEE

Code in 10 days

Day 8

dotSlash
Workshop series

Topics for Today

- Function Overloading
- Constructors and Destructors

Function Overloading

- A function name having several definitions that are differentiable by the number of or types of their arguments.
- The key to function overloading is it's argument list, also known as its *signature*.

Some examples,

```
void sqr(int a, float b);
void sqr(int x, float y);
```

} Same signature

```
void print(int i);
void print(char c);
```

} Different signatures

Interpretation

When a function is declared more than once, the compiler interprets the second declaration as follows:

- If the signatures of both functions match, the second declaration is treated as a re-declaration of the first.
- If the signatures of the two functions match, but the return types differ, the second function is treated as an erroneous declaration of the first, and flagged as an error.

Calling Overloaded Functions

- Overloaded functions are just like other functions.
- The number and type of arguments determines which function should be invoked.
- A function call first matches the prototypes available with the number and type of arguments provided in the call, and calls the appropriate function.

Example

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

void print(int i) {
cout << " Here is int " << i << endl;
}
void print(double f) {
cout << " Here is float " << f << endl;
}

int main() {
print(10);
print(10.10);
return 0;
}
```

```
Here is int 10
Here is float 10.1
```

Program 1

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

// function with 2 parameters
void display(int var1, double var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}

// function with double type single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}

// function with int type single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}

int main() {

    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);

    // call function with double type parameter
    display(b);

    // call function with 2 parameters
    display(a, b);

    return 0;
}
```

Constructors and Destructors

Constructors

- A constructor is a member function with the same name as its class.
- It is used to initialise objects of that class type with a legal initial value.
- If defined, it is called whenever a program creates an object of that class.
- They have no return type, not even void.
- Constructors are needed so that a compiler initialises an object as soon as it is created.

Types of Constructors

- Default Constructors
  - A constructor that accepts no parameters is called a default constructor.
- Parametrized Constructors
  - A constructor that can accept parameters for its invocation.
  - They are also called regular constructors.
  - When such a constructor is declared, the default constructor gets hidden
- Copy Constructor
  - A constructor of the form *classname(classname &)*, which is used by the compiler whenever we initialise an object using the values of another instance of the same type.

Default Constructors

The significance of default constructors is:

- We can create objects without having to type the initial values every time we create objects with prespecified values.
- We can also create an array of objects of that class type.

Default Constructor

```
// Program to illustrate the concept of
Constructors
#include <iostream>
using namespace std;

class construct
{
public:
        int a, b;

        // Default Constructor
        construct()
        {
                a = 10;
                b = 20;
        }
};

int main()
{
        // Default constructor called
automatically when the object is created
        construct c;
        cout << "a: " << c.a << endl
                << "b: " << c.b;
        return 1;
}
```

Parametrized Constructor

```
// Program to illustrate parameterized constructors
#include <iostream>
using namespace std;

class Point
{
  private:
            int x, y;
  public:
            // Parameterized Constructor
            Point(int x1, int y1)
            {
                        x = x1;
                        y = y1;
            }
            int getX()
            {
                        return x;
            }
            int getY()
            {
                        return y;
            }
};

int main()
{
            // Constructor called
            Point p1(10, 15);

            // Access values assigned by constructor
            cout << "p1.x = " << p1.getX() << ", p1.y = "
<< p1.getY();

            return 0;
}
```

Copy Constructors

A Copy Constructor may be called in the following cases:

1. When an object of the class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

3. When an object is constructed based on another object of the same class.

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; }
};
```

```
int main()
{
        Point p1(10, 15); // Normal constructor is called
        Point p2 = p1; // Copy constructor is called here
        cout << "p1.x = " << p1.getX() << ", p1.y = "
<< p1.getY();
        cout << "\np2.x = " << p2.getX() << ", p2.y =
" << p2.getY();

        return 0;
}
```

Calling Constructors

- Explicit Call

  This is when the name of the constructor is explicitly specified to invoke it, so that the object can be initialised.

  e.g. A obj1 = A(1, 2);

- Implicit Call

  This means that the constructor is called even when its name is not specified in the statement. When we create an object it gets called automatically.

  e.g. A obj1(13, 12);

Constructors & Inheritance

```
// Program to show the order of constructor calls in
Multiple Inheritance

#include <iostream>
using namespace std;

// first base class
class Parent1
{
    public:
   // first base class's Constructor
   Parent1()
   {
      cout << "Inside first base class" << endl;
   }
};
// second base class
class Parent2
{
   public:
  // second base class's Constructor
   Parent2()
   {
      cout << "Inside second base class" << endl;
   }
};
// child class inherits Parent1 and Parent2
class Child : public Parent1, public Parent2
{
                public:
                // child class's Constructor
                Child()
                {
                                cout << "Inside child class" <<
endl;
                }
};
// main function
int main() {

                // creating object of class Child
                Child obj1;
                return 0;
}
```

Constructor Overloading

```
// Program to illustrate Constructor overloading
#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;
}
```

Destructors

- A member function with the same name as that of a class, but it is preceded by a tilde(~) sign.

e.g. For a class Sample,
destructor would be ~Sample

- Destructors take no arguments, nor have return types.
- It is automatically invoked by the compiler when an object is destroyed.
- It cleans up the storage.
- Destructors are necessary to deallocate the resources allocated at the time of object creation.

Destructors

- Destructors cannot be overloaded.
- But they can be defined.

e.g.

```
~Sample()
{
        cout<<“Destructor\n”;
}
```

- They cannot be inherited.
- If a class has a destructor, each object of the class will be deinitialized before the object goes out of scope.

Program 2

```
#include<iostream>
using namespace std;
class Demo {
  private:
  int num1, num2;
  public:
  Demo(int n1, int n2) {
    cout<<"Inside Constructor"<<endl;
    num1 = n1;
    num2 = n2;
  }
  void display() {
    cout<<"num1 = "<< num1 <<endl;
    cout<<"num2 = "<< num2 <<endl;
  }
  ~Demo() {
      cout<<"Inside Destructor";
  }
};
int main() {
  Demo obj1(10, 20);
  obj1.display();
  return 0;
}
```

```
Inside Constructor
num1 = 10
num2 = 20
Inside Destructor
```

Thank You