Object Oriented Programming in C++

Segment-3

Course Code: 2301

Prepared by Sumaiya Deen Muhammad

Lecturer, Dept. of CSE, IIUC

Contents

Function Overloading:

1. Overloading function,
2. Constructor Overloading,
3. Copy constructor,
4. Default arguments,
5. Overloading ambiguity,
6. Address of overloaded function.

Overloading function

An *overloaded* function is actually a group of functions with the same name. When the function is called, which of them will be executed depends on the type and number of arguments supplied in the call.

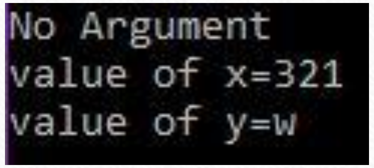
The following program contains three functions with the same name. There are three declarations, three function calls, and three function definitions. It uses the function signature—**the number of arguments**, and their **data-types** to distinguish one function from another.

```
#include<iostream>
using namespace std;
void method();
void method(int x);
void method(char y);

int main()
{
    int a;
    method();
    method(321);
    method('w');
    return 0;
}
```

```
void method()
{  cout<<"No Argument"<<endl;       }
void method(int x)
{  cout<<"value of  x="<<x<<endl;    }
void method(char y)
{    cout<<"value of  y="<<y<<endl;  }
```

Output



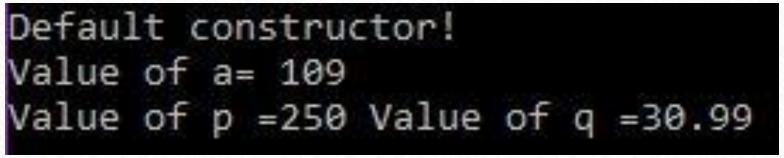
```
No Argument
value of x=321
value of y=w
```

Constructor Overloading

```
class Overloading
{
public:
 Overloading()
   { cout<<"Default constructor!"<<endl;}
   Overloading(int a)
   {
      cout<<"Value of  a= "<<a<<endl;
   }
   Overloading(int p, double q)
   {
      cout<<"Value of  p ="<<p<<" Value of  q ="<<q<<endl;
   }
};
```

```
int main()
{
   Overloading ob1, ob2(109), ob3(250, 30.99);
   return  0;
}
```

Output:



```
Default constructor!
Value of a= 109
Value of p =250 Value of q =30.99
```

Advantages of Overloading

1) The function can perform different operations and hence eliminates the use of different function names for the same kind of operations.
2) Program becomes easy to understand.
3) Easy maintainability of the code.
4) Function overloading brings flexibility in C++ programs.

Copy Constructor

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name. The compiler provides a default Copy Constructor to all the classes.

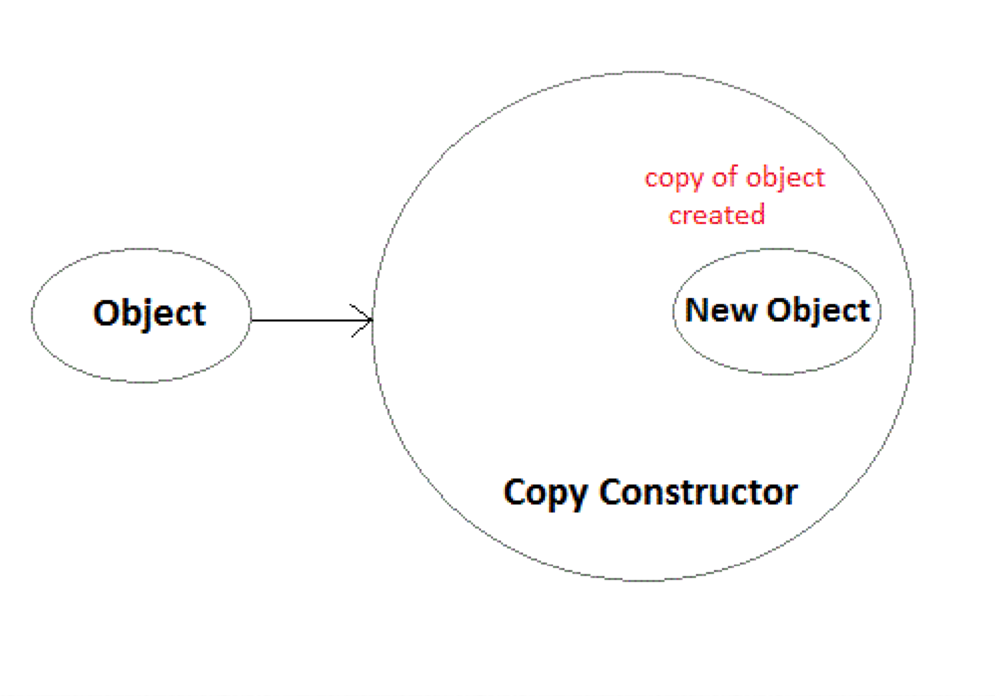
Syntax of Copy Constructor

```
Classname(const classname &objectname)
              { . . . . }
```

The copy constructor is used to –

1. Initialize one object from another of the same type.
2. Copy an object to pass it as an argument to a function.
3. Copy an object to return it from a function.

Copy Constructor (cont.)



Copy Constructor (cont.)

```
class Point
{private:
    int x, y;
public:
    Point(int x1, int y1)
    {     x = x1;
          y = y1;
    }
    Point(const Point &p2) // Copy constructor

    {     x = p2.x;
          y = p2.y;     }
    int getX()
          {  return x; }
    int getY()
       {  return y; }
};
```

```
int main()
{
    Point p1(10, 15); // Normal constructor is
called here
    Point p2 = p1; // Copy constructor is called
here

   // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = "
<< p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = "
<< p2.getY();

    return 0;
}
```

Default Arguments

A function can be called without specifying all its arguments. This won’t work on just any function. The function declaration must provide **default values** for those arguments that are not specified. In the following program the function test() takes two arguments. It’s called three times from main(). The first time it’s called with no arguments, the second time with one, and the third time with two. Why do the first two calls work? Because the called function provides default arguments, which will be used if the calling program doesn’t supply them. The default arguments are specified in the declaration for test():

```
void test(char=‘*’, int=45);    //declaration of default arguments
```

Default Arguments example

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

void test(char=’*’, int=45);        //declaration with default arguments

int main()
{
        test(); //prints 45 asterisks
        test(‘=’); //prints 45 equal signs
        test(‘+’, 30); //prints 30 plus signs
        return 0;
}
```

```
void test(char ch, int n)
{
        for(int j=0; j<n; j++) //loops n times
        cout << ch; //prints ch
        cout << endl;
}
```

Default Arguments example (cont.)

```
void Def_arg(int a=0, int b=0)
{
        cout<<“a= ”<<a<<“ b=”<<b<<endl;
}
int main()
{
        Def_arg();
        Def_arg(10);
        Def_arg(10,20);
        return 0;
}
```

Output:

```
a=0     b=0

a=10    b=0

a=10    b=10
```

Overloading Ambiguity

We know that compiler will decide which function to be invoked among the overloaded functions. When the compiler could not choose a function between two or more overloaded functions, the situation is called as an **ambiguity** in function overloading.

When the program contains ambiguity, the compiler will not compile the program.

The main cause of ambiguity in the program

- Type conversion
- Functions with default arguments
- Functions with pass by reference

Overloading Ambiguity Example

```
float f (float i)
{
        return i/2.0;
}
double f (double j)
{
        return j / 3.0;
}
int main()
{
        float x = 10.09;
        double y = 10.09;
        cout<<f(x);
        cout<<f(y);
        cout<<f(10);
        return 0;
}
```