CSE 232

Programming with C++

Lecture 9

Object Oriented Programming (2)

Prepared by

Md. Mijanur Rahman, Prof. Dr.
Dept. of Computer Science and Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh
Email: mijan@jkkniu.edu.bd
Web: www.mijanrahman.com

Contents

Object Oriented Programming

- C++ Polymorphism
- Function Overloading
- Operator Overloading
- Constructor Overloading
- Function Overriding
- Exception Handling

C++ 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.
- A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.
- Thus, the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Function Overloading

- When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading.
- Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.
- In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.

Function Overloading

- The parameters should follow any one or more than one of the following conditions for Function overloading:

- Parameters should have a different type
  - add(int a, int b)
  - add(double a, double b)

Function Overloading

- C++ Program Example:

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

void add(int a, int b)
{
  int c = a+b;
  cout << "sum = " << c << endl;
}

void add(double a, double b)
{
    double c = a+b;
    cout << "sum = " << c << endl;
}

// Driver code
int main()
{
    add(100, 250);
    add(25.30, 26.25);

    return 0;
}
```

```
sum = 350
sum = 51.55
```

Function Overloading

- C++ Program Example:

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

class Calculator {
public:
    // Overloaded max function for integers
    int max(int a, int b) {
        return (a > b) ? a : b;
    }
    // Overloaded max function for doubles
    double max(double a, double b) {
        return (a > b) ? a : b;
    }
    // Overloaded max function for characters
    char max(char a, char b) {
        return (a > b) ? a : b;
    }
};

int main() {
    Calculator calc;

    cout << "Max of 3 and 7 (int): " << calc.max(3, 7) << endl;
    cout << "Max of 4.5 and 2.3 (double): " << calc.max(4.5, 2.3) << endl;
    cout << "Max of 'a' and 'z' (char): " << calc.max('a', 'z') << endl;
    return 0;
}
```

```
Max of 3 and 7 (int): 7
Max of 4.5 and 2.3 (double): 4.5
Max of 'a' and 'z' (char): z
```

Operator Overloading

- C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands.
- Thus, a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Operator Overloading

- Example of Operator Overloading in C++

```
22 + i39
```

```cpp
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    Complex operator+(Complex obj)
    {
        Complex c;
        c.real = real + obj.real;
        c.imag = imag + obj.imag;
        return c;
    }
    void print() { cout << real << " + i" << imag << '\n'; }
};

int main()
{
    Complex c1(10, 25), c2(12, 14);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Operator Overloading

- Example:

```
Time t1: 1h 45m 50s
Time t2: 2h 20m 15s
Time t1 + t2: 4h 6m 5s
```

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

class Time {
private:
    int hours;
    int minutes;
    int seconds;

    void normalize() {
        if (seconds >= 60) {
            minutes += seconds / 60;
            seconds %= 60;
        }
        if (minutes >= 60) {
            hours += minutes / 60;
            minutes %= 60;
        }
    }

public:
    // Constructor
    Time(int h = 0, int m = 0, int s = 0){
        hours = h; minutes = m; seconds = s;
        normalize();
    }
    // Overloading the + operator to add two Time objects
    Time operator + (Time T) {
        Time R;
        R.seconds = seconds + T.seconds;
        R.minutes = minutes + T.minutes;
        R.hours = hours + T.hours;
        R.normalize();  // Ensure time is normalized
        return R;
    }

     // Display function
    void display() {
        cout << hours << "h " << minutes << "m " << seconds << "s" << endl;
    }
};

int main() {
    Time t1(1, 45, 50);  // 1 hour, 45 minutes, and 50 seconds
    Time t2(2, 20, 15);  // 2 hours, 20 minutes, and 15 seconds
    Time t3 = t1 + t2;   // Using overloaded + operator to add t1 and t2

    cout << "Time t1: ";
    t1.display();
    cout << "Time t2: ";
    t2.display();
    cout << "Time t1 + t2: ";
    t3.display();

   return 0;
}
```

Operator Overloading

- **Difference between Operator Functions and Normal Functions**

  - Operator functions are the same as normal functions.
  - The only differences are, that the name of an operator function is always the **operator** keyword followed by **the symbol of the operator**, and operator functions are called when the corresponding operator is used.

Constructor Overloading

- In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.
  - 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.

Constructor Overloading

- C++ Program Example:

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

class Rectangle {
private:
    int width;
    int height;

public:
    // Default constructor
    Rectangle(){
        width = height = 0;
    }

    // Constructor with one parameter (for square)
    Rectangle(int side){
        width = height = side;
    }

    // Constructor with two parameters
    Rectangle(int w, int h){
        width = w; height = h;
    }

    // Method to calculate area
    int area() {
        return width * height;
    }
    // Method to display dimensions
    void display() {
        cout << "Width: " << width << ", Height: " << height << endl;
    }
};

int main() {
    Rectangle rect1;        // Calls default constructor
    Rectangle rect2(5);     // Calls single-parameter constructor (square)
    Rectangle rect3(4, 6);  // Calls two-parameter constructor

    cout << "Rectangle 1: ";
    rect1.display();
    cout << "Area: " << rect1.area() << endl;

    cout << "Rectangle 2: ";
    rect2.display();
    cout << "Area: " << rect2.area() << endl;

    cout << "Rectangle 3: ";
    rect3.display();
    cout << "Area: " << rect3.area() << endl;

    return 0;
}
```

```
Rectangle 1: Width: 0, Height: 0
Area: 0
Rectangle 2: Width: 5, Height: 5
Area: 25
Rectangle 3: Width: 4, Height: 6
Area: 24
```

Function Overriding

- Function overriding is a type of polymorphism in which we redefine the member function of a class which it inherited from its base class.
- The function signature remains same but the working of the function is altered to meet the needs of the derived class.
- So, when we call the function using its name for the parent object, the parent class function is executed. But when we call the function using the child object, the child class version will be executed.

Function Overriding

- C++ Example

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

class Parent {
public:
    void display()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void display()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Parent P;
    P.display();
    Child C;
    C.display();
    return 0;
}
```

```
Base Function
Derived Function
```

Exception Handling

- An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).
- In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. The process of handling these exceptions is called exception handling.
- Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.

Exception Handling

- **C++ try and catch**
- C++ provides an inbuilt feature for Exception Handling. It can be done using the following specialized keywords: try, catch, and throw with each having a different purpose.
- **Syntax of try-catch in C++**

```
try {
    // Code that might throw an exception
    throw SomeExceptionType("Error message");
 }
catch( ExceptionName e1 )  {
    // catch block catches the exception that is thrown from try block
 }
```

Exception Handling

- Examples of Exception Handling in C++

```
Exception: Division by zero not allowed!
```

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

int main()
{
    try {
        int numerator = 10;
        int denominator = 0;
        int res;

        // check if denominator is 0 then throw error.
        if (denominator == 0) {
            throw runtime_error("Division by zero not allowed!");
        }

        res = numerator / denominator;
        cout << "Result after division: " << res << endl;
    }
    catch (const exception& e) {
        // print the exception
        cout << "Exception: " << e.what() << endl;
    }

    return 0;
}
```

Lecture 9

Object Oriented Programming (2)

THE END