Lab 9

Items	Description
Course Title	Object Oriented Programming
Lab Title	Classes (Operator Overloading)
Duration	3 Hours
Tools	Eclipse/ C++
Objective	To get familiar with the use of different concepts in classes in c++

Operator Overloading

Operator overloading in C++ allows you to redefine how operators like +, -, ==, [], () and others behave for user-defined types (classes). This enhances code readability and functionality by allowing operators to work naturally with objects, just like they do with built-in data types.

Types of Operator Overloading

1. **Binary Operators**: Operators that work with two operands, like +, -, *, etc.
2. **Relational Operators**: Comparison operators like ==, !=, <, >, used to compare two objects.
3. **Unary Operators**: Operators that work with a single operand, like - (negation), ++, --.
4. **Subscript Operator**: Overloads the [] operator to access elements in an array-like structure.
5. **Function Call Operator (Parenthesis)**: Overloads the () operator to make an object behave like a function.

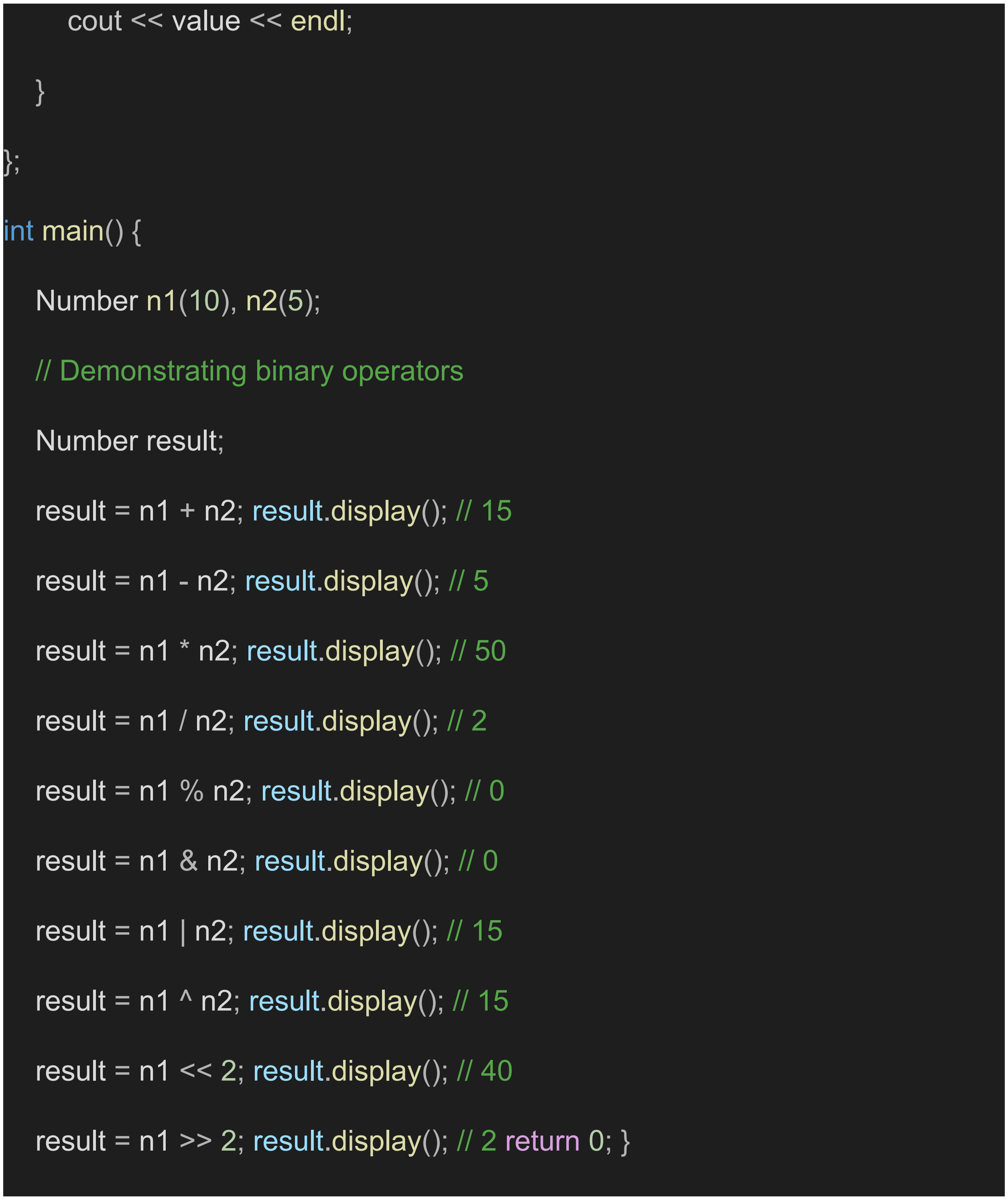
1. Binary Operators Overloading (All Binary Operators)

Binary operators in C++ include +, -, *, /, %, &, |, ^, <<, >>. Here's a class example where all these binary operators are overloaded for a Number class:

```
#include <iostream>

using namespace std;

class Number {

    int value;

public:

    Number(int v = 0) : value(v) {}

    // Overloading + operator

    Number operator+(const Number& other) {

        return Number(value + other.value);

    }

    // Overloading - operator

    Number operator-(const Number& other) {

        return Number(value - other.value);

    }



    // Overloading * operator
```

```
Number operator*(const Number& other) {

    return Number(value * other.value);

}

// Overloading / operator

Number operator/(const Number& other) {

    return Number(value / other.value);

}

// Overloading % operator

Number operator%(const Number& other) {

    return Number(value % other.value);

}

// Overloading & operator (bitwise AND)

Number operator&(const Number& other) {

    return Number(value & other.value);

}



// Overloading | operator (bitwise OR)
```

```
Number operator|(const Number& other) {

    return Number(value | other.value);

}

// Overloading ^ operator (bitwise XOR)

Number operator^(const Number& other) {

    return Number(value ^ other.value);

}

// Overloading << operator (bitwise left shift)

Number operator<<(int shift) {

    return Number(value << shift);

}

// Overloading >> operator (bitwise right shift)

Number operator>>(int shift) {

    return Number(value >> shift);

}



void display() const {
```



```
        cout << value << endl;
    }
};
int main() {
    Number n1(10), n2(5);
    // Demonstrating binary operators
    Number result;
    result = n1 + n2; result.display(); // 15
    result = n1 - n2; result.display(); // 5
    result = n1 * n2; result.display(); // 50
    result = n1 / n2; result.display(); // 2
    result = n1 % n2; result.display(); // 0
    result = n1 & n2; result.display(); // 0
    result = n1 | n2; result.display(); // 15
    result = n1 ^ n2; result.display(); // 15
    result = n1 << 2; result.display(); // 40
    result = n1 >> 2; result.display(); // 2 return 0; }
```

2. Relational Operators Overloading (All Relational Operators)

Relational operators compare two operands and include ==, !=, <, >, <=, and >=. Here's an example where these operators are overloaded for a Number class:

```
#include <iostream>

using namespace std;

class Number {

    int value;

public:

    Number(int v = 0) : value(v) {}

    // Overloading == operator

    bool operator==(const Number& other) const {

        return value == other.value;

    }

    // Overloading != operator

    bool operator!=(const Number& other) const {

        return value != other.value;

    }

    // Overloading < operator

    bool operator<(const Number& other) const {
        return value < other.value;
    }
    // Overloading > operator
    bool operator>(const Number& other) const {
        return value > other.value;
    }
    // Overloading <= operator
    bool operator<=(const Number& other) const {
        return value <= other.value;
    }
    // Overloading >= operator
    bool operator>=(const Number& other) const {
        return value >= other.value;
    }
};
int main() {
    Number n1(10), n2(5);
    // Demonstrating relational operators

    if (n1 == n2) cout << "n1 equals n2" << endl;

    if (n1 != n2) cout << "n1 is not equal to n2" << endl;

    if (n1 > n2) cout << "n1 is greater than n2" << endl;

    if (n1 < n2) cout << "n1 is less than n2" << endl;

    if (n1 >= n2) cout << "n1 is greater than or equal to n2" << endl;

    if (n1 <= n2) cout << "n1 is less than or equal to n2" << endl;

    return 0;

}
```

3. Unary Operators Overloading (All Unary Operators)

Unary operators operate on a single operand and include +, -, ++, --, and the logical negation operator !. Here's an example that overloads these operators for a Number class:

```
#include <iostream>

using namespace std;


class Number {

    int value;

public:

    Number(int v = 0) : value(v) {}

    // Overloading unary - operator (negation)

    Number operator-() {

        return Number(-value);

    }

    // Overloading unary ++ operator (pre-increment)

    Number& operator++() {

        ++value;

        return *this;

    }
```

```
// Overloading unary ++ operator (post-increment)

Number operator++(int) {

    Number temp = *this;

    ++value;

    return temp;

}

// Overloading unary -- operator (pre-decrement)

Number& operator--() {

    --value;

    return *this;

}

// Overloading unary -- operator (post-decrement)

Number operator--(int) {

    Number temp = *this;

    --value;

    return temp;

}
```

```
    // Overloading logical not (!) operator
    bool operator!() const {
        return value == 0;
    }
    void display() const {
        cout << value << endl;
    }
};
int main() {
    Number n(10);
    -n; n.display(); // -10
    ++n; n.display(); // 11 (pre-increment)
    n++; n.display(); // 12 (post-increment)
    --n; n.display(); // 11 (pre-decrement)
    n--; n.display(); // 10 (post-decrement)
    if (!n) cout << "n is zero" << endl;
    else cout << "n is not zero" << endl;
    return 0;  }
```

National University
of computer and emerging sciences

4. Subscript Operator Overloading ([ ])

The subscript operator `[ ]` allows array-like indexing. Here's an example that overloads this operator for a custom `Array` class:

```
#include <iostream>

using namespace std;

class Array {

    int arr[5];

public:

    Array() {

        for (int i = 0; i < 5; ++i)

            arr[i] = i + 1; // Initialize array with values 1, 2, 3, 4, 5

    }

    // Overloading the [] operator

    int& operator[](int index) {

        if (index >= 0 && index < 5)

            return arr[index];

        else

            cout << "Index out of bounds!" << endl;

        exit(0);
    }

    void display() {

        for (int i = 0; i < 5; i++) {

            cout << arr[i] << " ";

        }

        cout << endl;

    }

};

int main() {

    Array a;

    cout << "Element at index 2: " << a[2] << endl; // Output: 3

    a[2] = 10;

    cout << "Updated element at index 2: " << a[2] << endl; // Output: 10

    a.display(); // Display updated array: 1 2 10 4 5


    return 0;

}
```

5. Function Call (Parenthesis) Operator Overloading (`()`)

The parenthesis `()` operator allows an object to be called like a function. Here's an example that overloads this operator for a `Multiplier` class:

```
#include <iostream>

using namespace std;

class Multiplier {

    int factor;

public:

    Multiplier(int f) : factor(f) {}

    // Overloading () operator to behave like a function

    int operator()(int x) {

        return factor * x;

    }

};

int main() {

    Multiplier multiplyBy2(2);

    cout << "Result: " << multiplyBy2(10) << endl; // Output: 20

    cout << "Result: " << multiplyBy2(5) << endl;  // Output: 10

    return 0; }
```

National University of computer and emerging sciences

Friend Function

A **friend function** is a function that is not a member of a class but can access the class's private and protected members. This is useful when you want to allow certain external functions or other classes to access private data while keeping it hidden from other functions.

Syntax of Friend Function

- To declare a function as a friend, you use the keyword `friend` inside the class where the function needs access to private members.

```
#include <iostream>

using namespace std;

class Box {

    int length;  // Private member

public:

    Box(int l = 0) : length(l) {}  // Constructor

    // Declare the friend function

    friend int getLength(Box);  // A friend function that can access private
members

};

// Friend function definition

int getLength(Box b) {

    // Accessing the private member 'length' from the class Box

    return b.length;
}

int main() {

    Box box1(10);  // Create object of Box with length = 10

    // Using the friend function to get the length of the box

    cout << "Length of the box: " << getLength(box1) << endl;  // Output: Length
of the box: 10

    return 0;

}
```

Header File Implementation

1. Box.h (Header File)

```
// Box.h
#ifndef BOX_H
#define BOX_H

#include <iostream>
using namespace std;

class Box {
private:
    int length;

public:
    Box(int l = 0);  // Constructor
    // Overloading the + operator as a member function
    Box operator+(const Box& other) const;
    // Declare display function as a friend function
    friend void display(const Box& b);
};

#endif
```

2. Box.cpp (Class Implementation File)

```
// Box.cpp
#include "Box.h"

// Constructor
Box::Box(int l) : length(l) {}

// Member function to overload + operator
Box Box::operator+(const Box& other) const {
    // Add the lengths of two Box objects
    return Box(this->length + other.length);
}

// Friend function definition to display the length
void display(const Box& b) {
    cout << "Length: " << b.length << endl;
}
```

3. main.cpp (Main Program File)

```
// main.cpp
#include "Box.h"

int main() {
    Box box1(10);  // Box with length 10
    Box box2(20);  // Box with length 20

    // Use the overloaded + operator (member function)
    Box box3 = box1 + box2;

    // Use the friend function to display the lengths of all boxes
    cout << "Box 1: "; display(box1);
    cout << "Box 2: "; display(box2);
    cout << "Box 3 (Box 1 + Box 2): "; display(box3);

    return 0;
}
```

National University
of computer and emerging sciences

Lab Task

Note: Submission guidelines for this task are the same as for the lab task 6. Check out lab task 6.

Task1: Simulate the Integer Data Type in C++

Create 3 files Integer.h, Interger.cpp and main.cpp.

You are required to create a class named `Integer` that simulates the behavior of the integer data type in C++. The class will have one integer data member and provide the following functionalities:

Class Requirements:

1. **Data Member:**
   - One private integer data member (`value`) to store the integer.
2. **Constructors:**
   - A **default constructor** to initialize the integer to `0`.
   - A **parameterized constructor** to initialize the integer to a specific value.
3. **Getter/Setter Methods:**
   - A `get()` method to return the value of the integer.
   - A `set()` method to set the value of the integer.
4. **Operator Overloading:** Implement the following operator overloads:
   - **Unary Operators:**
     - `++` (both **prefix** and **postfix**) to increment the integer.
     - `--` (both **prefix** and **postfix**) to decrement the integer.
   - **Binary Arithmetic Operators:**
     - `+` to add two `Integer` objects.
     - `-` to subtract one `Integer` object from another.
     - `*` to multiply two `Integer` objects.
     - `/` to divide one `Integer` object by another (handle division by zero by throwing an exception).
5. **Display Method:**
   - Implement a friend function `display()` that prints the value of the integer.

National University of computer and emerging sciences

Task2: Model Complex Numbers and Overload Operators

Create 3 files Complex.h, Complex.cpp and main.cpp.

You are required to create a class named `Complex` to represent complex numbers. The class will have two data members: real and imaginary parts.You will implement several constructors, getter and setter methods, and overload operators as specified below.

Class Requirements:

1. **Data Members**:
    - `double real`: Represents the real part of the complex number.
    - `double imaginary`: Represents the imaginary part of the complex number.
2. **Constructors**:
    - A **constructor with default parameters** to initialize the real and imaginary parts of the complex number.
3. **Getter and Setter Methods**:
    - `void setReal(double r)`: Sets the value of the real part.
    - `double getReal() const`: Returns the value of the real part.
    - `void setImaginary(double i)`: Sets the value of the imaginary part.
    - `double getImaginary() const`: Returns the value of the imaginary part.
4. **Operator Overloading**: You are required to overload the following operators for the `Complex` class:
    - **Binary Operators**:
        - **+ operator**: Add two complex numbers.
        - **- operator**: Subtract a complex number passed as an argument from the caller object.
        - *** operator**: Multiply two complex numbers.
        - **= operator**: Overload the assignment operator for complex numbers.
    - **Unary Operator**:
        - **! operator**: Return `true` if both the real and imaginary parts of the complex number are zero, otherwise return `false`.

Task3: Overload Subscript and Function Call Operators

Create 3 files Array.h, Array.cpp and Array.cpp.

You are required to create a class `Array` that simulates a dynamic array of integers, with additional functionality for performing operations using the function call operator. You will implement constructors, getter and setter methods, and overload the subscript (`[]`) and function call (`()`) operators as specified below.

Class Requirements:

1. **Data Members**:
   - `int* data`: A pointer to dynamically allocated memory for storing integer elements.
   - `int size`: The size of the array.
2. **Constructors**:
   - A **constructor** that allocates memory for the array of the specified size.
   - A **copy constructor** for deep copying the array.
   - A **destructor** to deallocate the memory.
3. **Getter Method**:
   - `int getSize() const`: Returns the size of the array.
4. **Operator Overloading**: You are required to overload the following operators for the class:
   - **Subscript Operator (`[]`)**:
     - The `[]` operator should allow access to elements of the array.
     - Overload it for **both read and write** access to the elements.
   - **Function Call Operator (`()`)**:
     - Overload the function call operator to calculate the sum of all elements in the array.
     - If a **single integer argument** is passed, return the value at the corresponding index (similar to `[]` but safer).
     - If **no argument** is passed, return the sum of the array elements.