CC-211L

Object Oriented Programming

Laboratory 07

Operator Overloading

Version: 1.0.0

Release Date: 02-03-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Overloading Unary Operators
- Overloading ++ Operator
- Overloading – Operator
- Dynamic Memory Management
- Operators as Members vs Non-Members
- Conversion between Types
- Overloading the Function call Operator

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Azka Saddiqa	azka.saddiqa@pucit.edu.pk
Teacher Assistants	Saad Rahman	bsef19m021@pucit.edu.pk
	Zain Ali Shan	bcsf19a022@pucit.edu.pk

Background and Overview:

Operator Overloading:

In C++, we can make operators work for user-defined classes. This means 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 overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

© https://www.geeksforgeeks.org/operator-overloading-c/

Unary Operators:

Unary Operator is an operation that is performed on only one operand. Unary Operators contrast with the Binary operators, which use two operands to calculate the result. Unary operators can be used either in the prefix position or the postfix position of the operand. To perform operations using Unary Operators, we need to use one single operand. There are various types of unary operators, and all these types are of equal precedence, having right-to-left associativity.

Dynamic Memory Management:

Dynamic memory management in C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack.

Type Conversion:

Type conversion is the process that converts the predefined data type of one variable into an appropriate data type. The main idea behind type conversion is to convert two different data type variables into a single data type to solve mathematical and logical expressions easily without any data loss.

Activities:

Pre-Lab Activities:

Overloading Unary Operators:

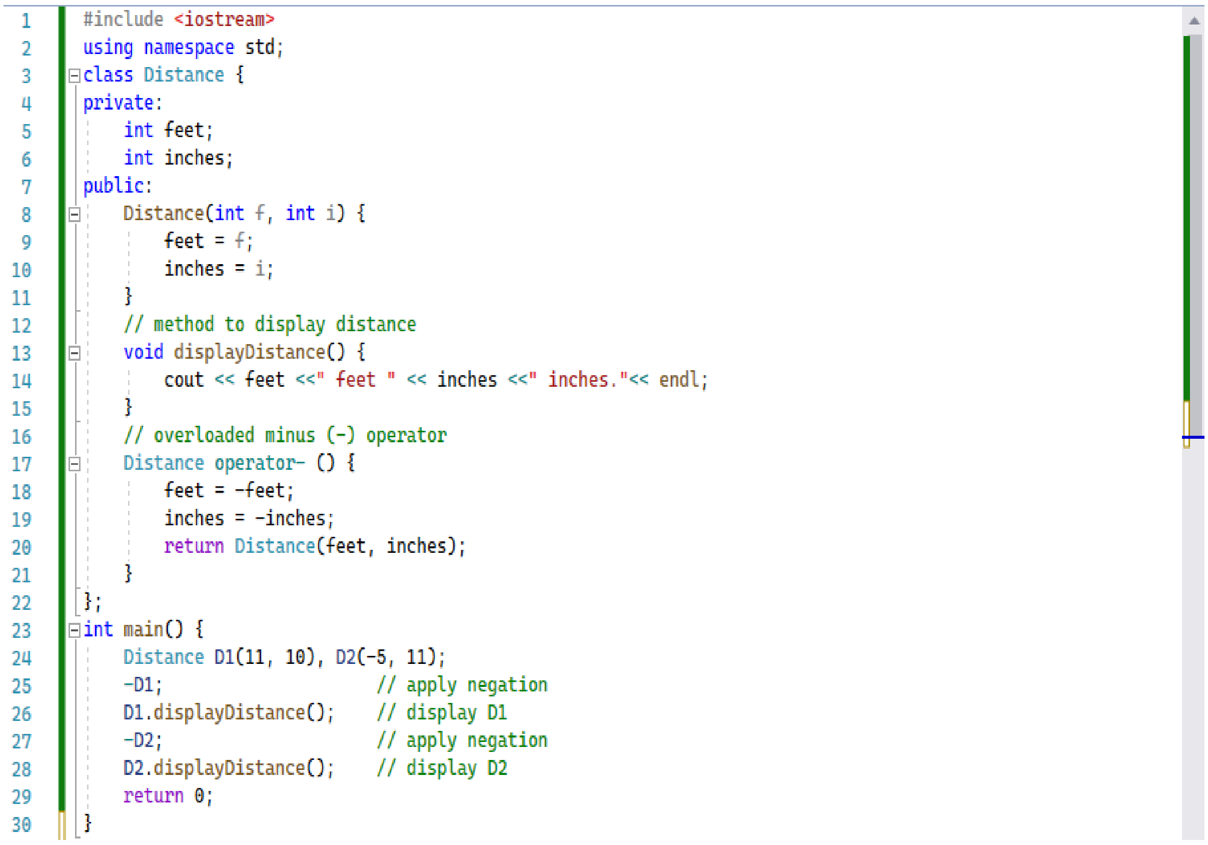
The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators
- The unary minus (-) operator
- The logical not (!) operator

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometimes they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

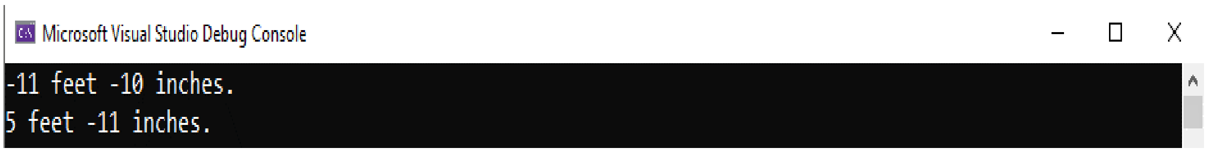
Example:



```cpp
#include <iostream>
using namespace std;
class Distance {
private:
    int feet;
    int inches;
public:
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    // method to display distance
    void displayDistance() {
        cout << feet <<" feet " << inches <<" inches."<< endl;
    }
    // overloaded minus (-) operator
    Distance operator- () {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
    }
};
int main() {
    Distance D1(11, 10), D2(-5, 11);
    -D1;                     // apply negation
    D1.displayDistance();    // display D1
    -D2;                     // apply negation
    D2.displayDistance();    // display D2
    return 0;
}
```

Fig. 01 (Overloading (-) Operator)

Output:



Microsoft Visual Studio Debug Console

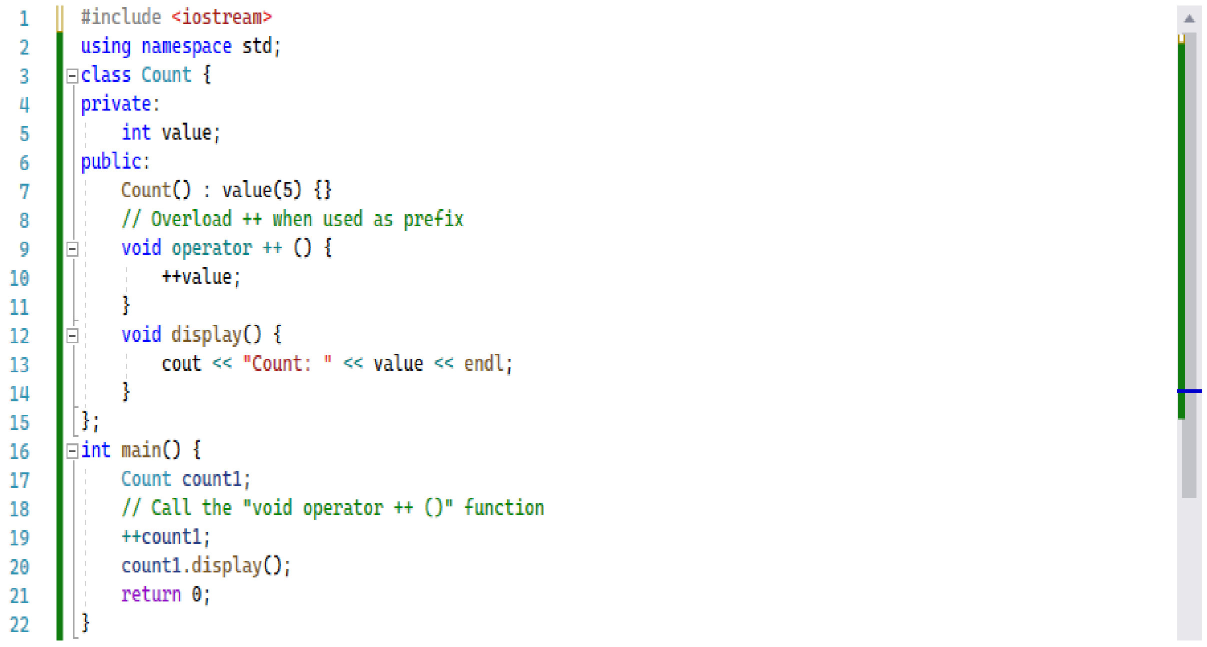
```
-11 feet -10 inches.
5 feet -11 inches.
```

Fig. 02 (Overloading (-) Operator)

Overloading Increment and Decrement Operators:

Following example explain how Increment (++) operator can be overloaded for prefix usage.

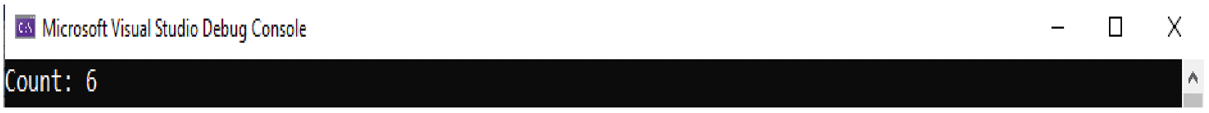
Example:



```cpp
#include <iostream>
using namespace std;
class Count {
private:
    int value;
public:
    Count() : value(5) {}
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
    void display() {
        cout << "Count: " << value << endl;
    }
};
int main() {
    Count count1;
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```

Fig. 03 (Overloading ++ Operator)

Output:



Microsoft Visual Studio Debug Console

```
Count: 6
```

Fig. 04 (Overloading ++ Operator)

In the above example, when we use ++count1, the void operator ++ () is called. This increases the value attribute for the object count1 by 1. It works only when ++ is used as a prefix. To make ++ work as a postfix we use this syntax.

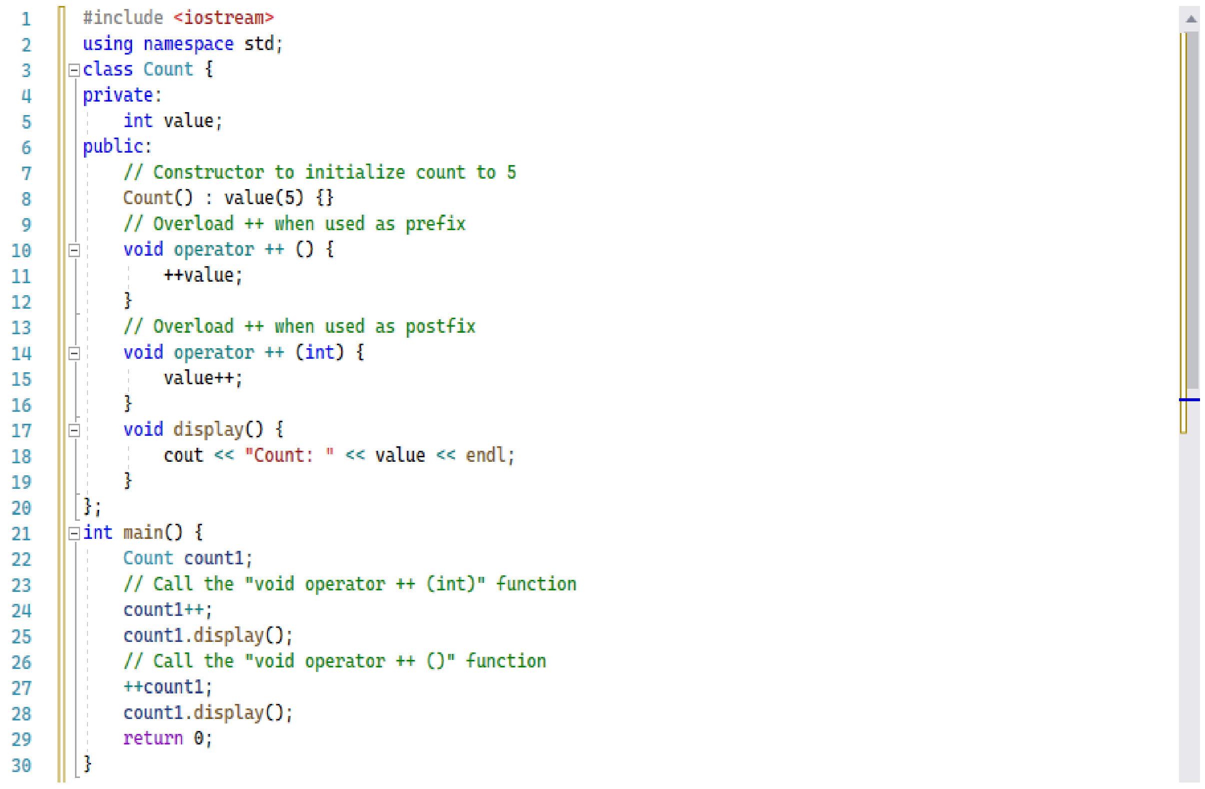
void operator ++ (int){

//code

}

Notice the int inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

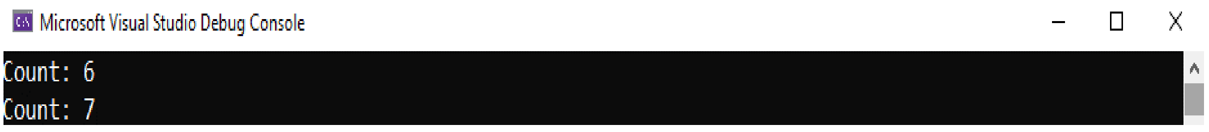
Example:



```cpp
#include <iostream>
using namespace std;
class Count {
private:
    int value;
public:
    // Constructor to initialize count to 5
    Count() : value(5) {}
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
    // Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }
    void display() {
        cout << "Count: " << value << endl;
    }
};
int main() {
    Count count1;
    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```

Fig. 05 (Overloading ++ Operator)

Output:



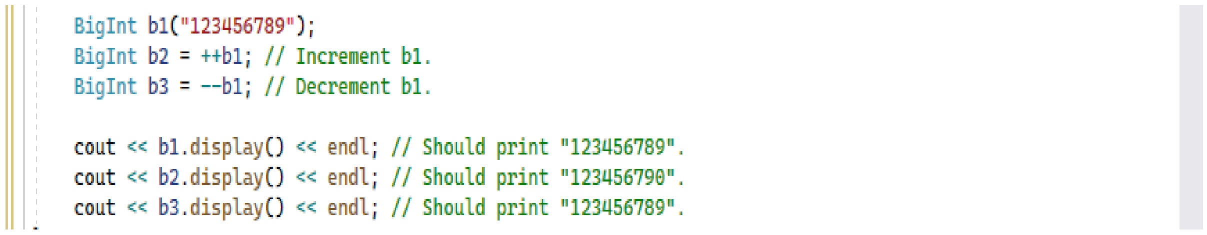
```
Count: 6
Count: 7
```

Fig. 06 (Overloading ++ Operator)

Task 01: Big Integer **[Estimated time 30 minutes / 20 marks]**

- Write a C++ class called BigInt that represents a large integer. The BigInt class should overload the unary ++ (pre-increment) and -- (pre-decrement) operators.
- The BigInt class should be able to represent integers of arbitrary size. You can do this by storing the digits of the integer as an array of integers, where each element in the array represents a single digit. For example, the integer 12345 would be represented as the array {5, 4, 3, 2, 1}.
- When the ++ operator is called, it should increment the BigInt object's integer value by 1 and return the updated BigInt object.
- When the -- operator is called, it should decrement the BigInt object's integer value by 1 and return the updated BigInt object.

Here's an example of how the BigInt class should be used:



```
BigInt b1("123456789");
BigInt b2 = ++b1; // Increment b1.
BigInt b3 = --b1; // Decrement b1.

cout << b1.display() << endl; // Should print "123456789".
cout << b2.display() << endl; // Should print "123456790".
cout << b3.display() << endl; // Should print "123456789".
```

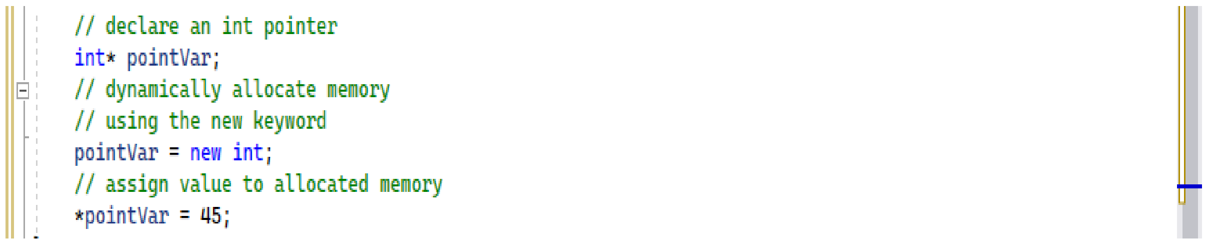
Fig. 07 (Pre-Lab Task)

In-Lab Activities:

Dynamic Memory Management:

C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.
In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.
In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.
We can allocate and then deallocate memory dynamically using the new and delete operators respectively.

new Operator:
The new operator allocates memory to a variable.

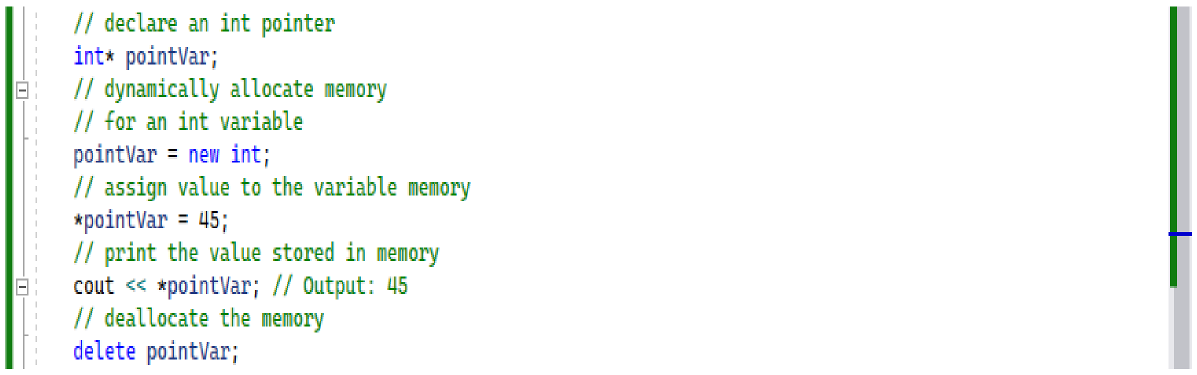


```
// declare an int pointer
int* pointVar;
// dynamically allocate memory
// using the new keyword
pointVar = new int;
// assign value to allocated memory
*pointVar = 45;
```

Fig. 08 (new Operator)

Here, we have dynamically allocated memory for an int variable using the new operator.
Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

delete Operator:
The delete operator deallocates memory assigned to a variable.

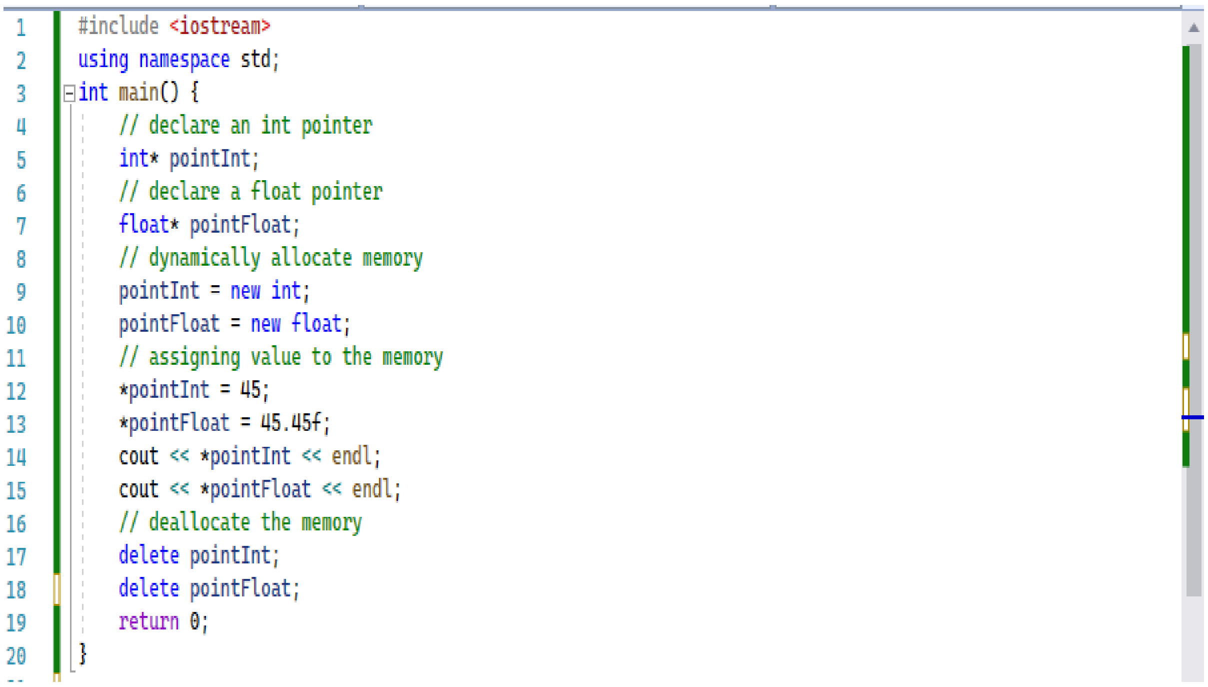


```
// declare an int pointer
int* pointVar;
// dynamically allocate memory
// for an int variable
pointVar = new int;
// assign value to the variable memory
*pointVar = 45;
// print the value stored in memory
cout << *pointVar; // Output: 45
// deallocate the memory
delete pointVar;
```

Fig. 09 (delete Operator)

Here, we have dynamically allocated memory for an int variable using the pointer pointVar.
After printing the contents of pointVar, we deallocated the memory using delete.

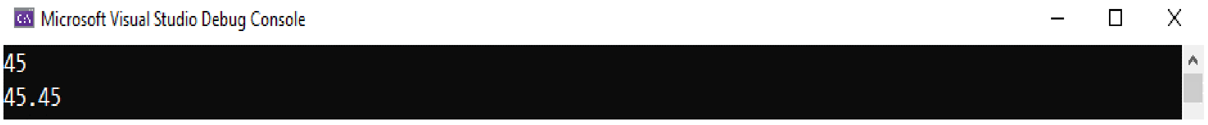
Example:



```cpp
#include <iostream>
using namespace std;
int main() {
    // declare an int pointer
    int* pointInt;
    // declare a float pointer
    float* pointFloat;
    // dynamically allocate memory
    pointInt = new int;
    pointFloat = new float;
    // assigning value to the memory
    *pointInt = 45;
    *pointFloat = 45.45f;
    cout << *pointInt << endl;
    cout << *pointFloat << endl;
    // deallocate the memory
    delete pointInt;
    delete pointFloat;
    return 0;
}
```

Fig. 10 (Dynamic Memory Allocation)

Output:



```
45
45.45
```

Fig. 11 (Dynamic Memory Allocation)

Conversion between types:

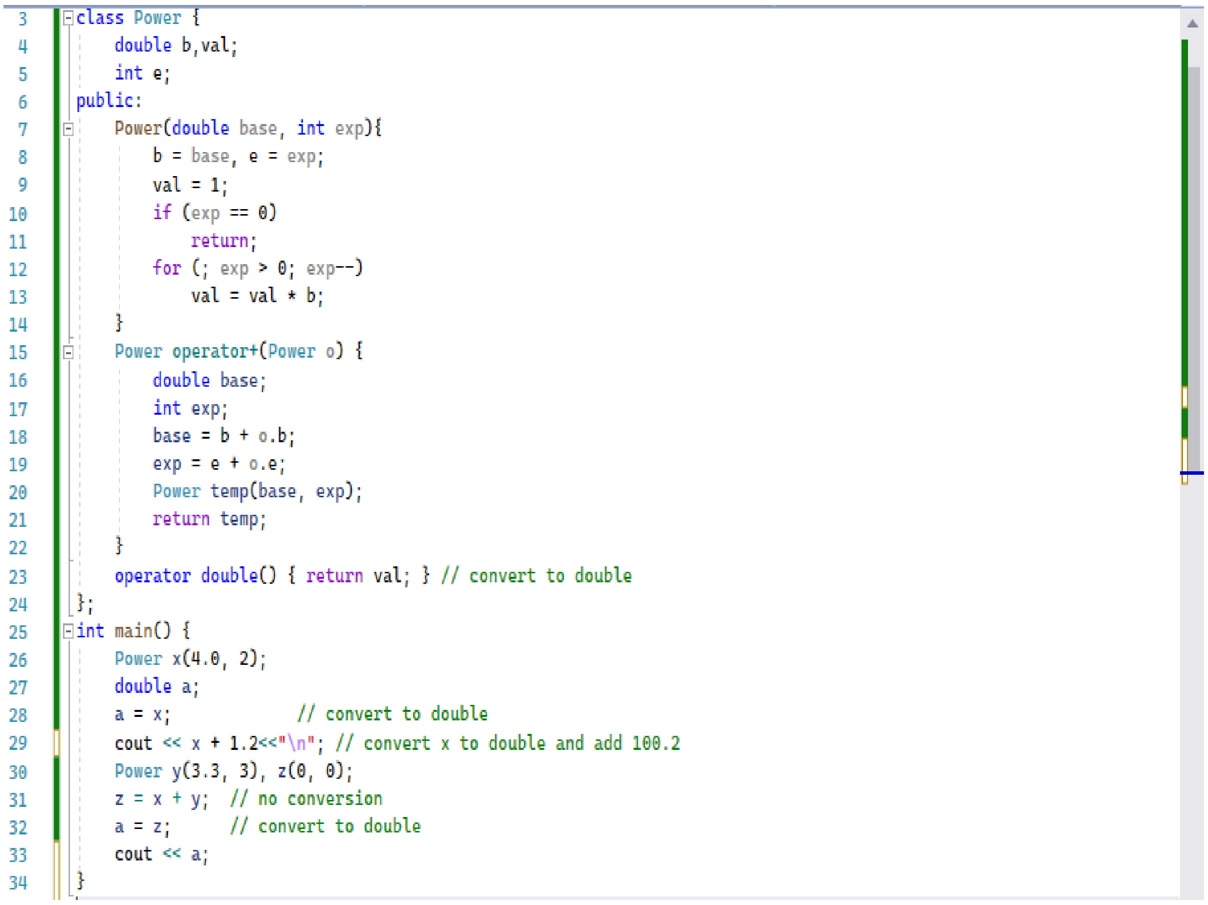
A conversion operator (also called a cast operator) also can be used to convert an object of one class to another type. Such a conversion operator must be a non-static member function. The function prototype:

MyClass::operator string() const;

declares an overloaded cast operator function for converting an object of class MyClass into a temporary string object. The operator function is declared const because it does not modify the original object. The return type of an overloaded cast operator function is implicitly the type to which the object is being converted.

Overloaded cast operator functions can be defined to convert objects of user-defined types into fundamental types or into objects of other user-defined types.

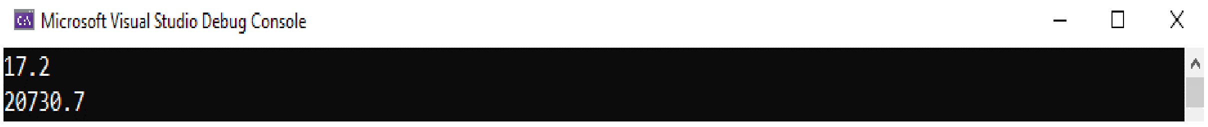
Example:



```
class Power {
    double b,val;
    int e;
public:
    Power(double base, int exp){
        b = base, e = exp;
        val = 1;
        if (exp == 0)
            return;
        for (; exp > 0; exp--)
            val = val * b;
    }
    Power operator+(Power o) {
        double base;
        int exp;
        base = b + o.b;
        exp = e + o.e;
        Power temp(base, exp);
        return temp;
    }
    operator double() { return val; } // convert to double
};
int main() {
    Power x(4.0, 2);
    double a;
    a = x;              // convert to double
    cout << x + 1.2<<"\n"; // convert x to double and add 100.2
    Power y(3.3, 3), z(0, 0);
    z = x + y;  // no conversion
    a = z;      // convert to double
    cout << a;
}
```

Fig. 12 (Conversion between types)

Output:



Microsoft Visual Studio Debug Console

```
17.2
20730.7
```

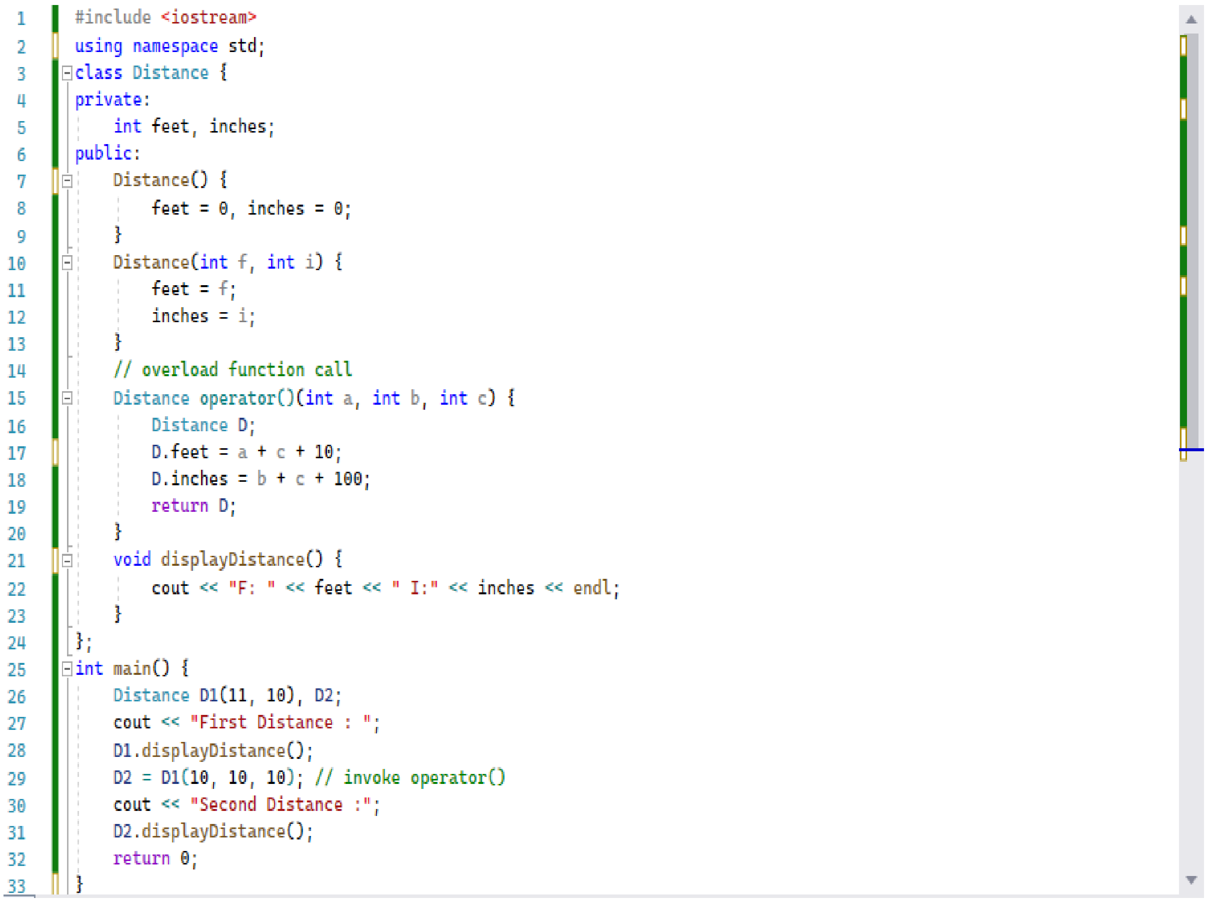
Fig. 13 (Conversion between types)

Overloading the Function Call Operator:

Overloading the function call operator () is powerful, because functions can take an arbitrary number of comma-separated parameters. In a customized String class, for example, you could overload this operator to select a substring from a String—the operator's two integer parameters could specify the start location and the length of the substring to be selected. The operator() function could check for such errors as a start location out of range or a negative substring length.

When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

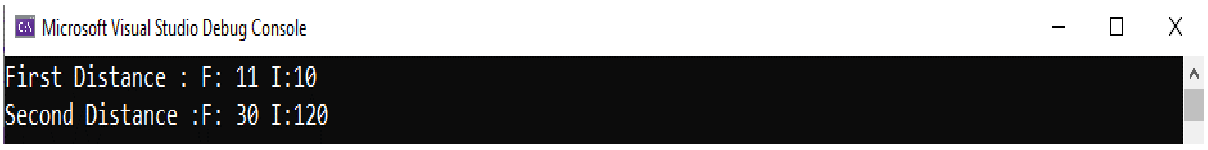
Example:



```
#include <iostream>
using namespace std;
class Distance {
private:
    int feet, inches;
public:
    Distance() {
        feet = 0, inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    // overload function call
    Distance operator()(int a, int b, int c) {
        Distance D;
        D.feet = a + c + 10;
        D.inches = b + c + 100;
        return D;
    }
    void displayDistance() {
        cout << "F: " << feet << " I:" << inches << endl;
    }
};
int main() {
    Distance D1(11, 10), D2;
    cout << "First Distance : ";
    D1.displayDistance();
    D2 = D1(10, 10, 10); // invoke operator()
    cout << "Second Distance :";
    D2.displayDistance();
    return 0;
}
```

Fig. 14 (Overload Function call operator)

Output:



```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

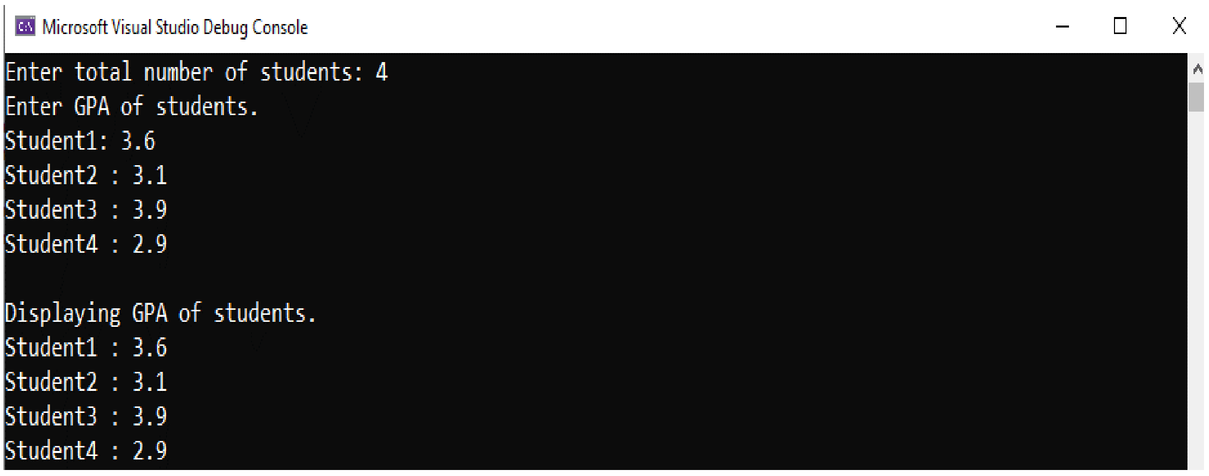
Fig. 15 (Overload Function call operator)

Task 01: Students GPA **[Estimated time 30 minutes / 20 marks]**

Create a C++ program which:

- Asks user to enter total number of students
- Store marks of all students in an array
- Release the memory of array at the end of the program

Sample Output:



```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2 : 3.1
Student3 : 3.9
Student4 : 2.9

Displaying GPA of students.
Student1 : 3.6
Student2 : 3.1
Student3 : 3.9
Student4 : 2.9
```

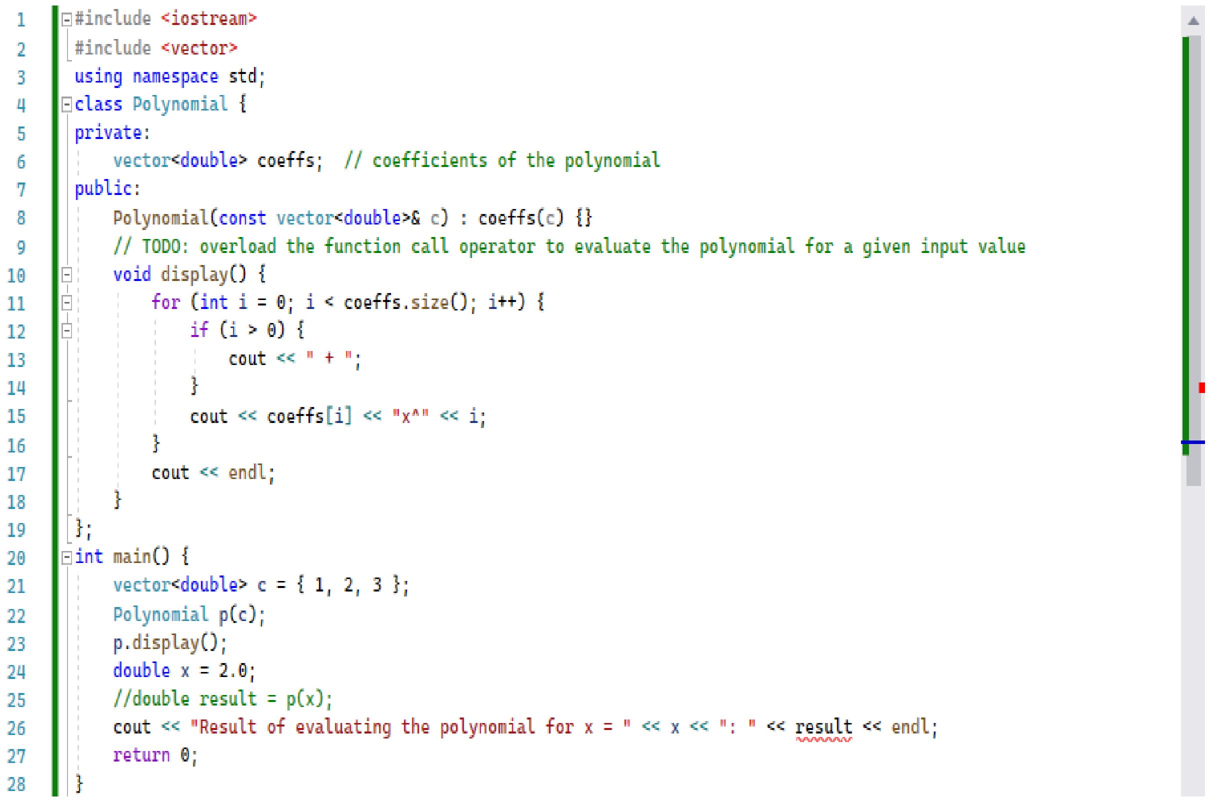
Fig. 16 (In-Lab Task)

Task 02: Cast Complex Numbers **[Estimated time 30 minutes / 30 marks]**

- Write a C++ program that defines a Complex class representing a complex number
- Overloads the cast operator to convert a Complex object to a string in the format "a + bi", where a and b are the real and imaginary parts of the complex number, respectively.
- The program should then create a Complex object, convert it to a string, and output the result to the console.

Task 03: Overloading Polynomials **[Estimated time 35 minutes / 30 marks]**

- Write a C++ program that defines a Polynomial class representing a polynomial function of a single variable
- Overloads the function call operator to evaluate the polynomial for a given value of its input variable.
- The program should then create a Polynomial object, evaluate it for a specific input value, and output the result to the console.
- Complete the code given below



```cpp
#include <iostream>
#include <vector>
using namespace std;
class Polynomial {
private:
    vector<double> coeffs;  // coefficients of the polynomial
public:
    Polynomial(const vector<double>& c) : coeffs(c) {}
    // TODO: overload the function call operator to evaluate the polynomial for a given input value
    void display() {
        for (int i = 0; i < coeffs.size(); i++) {
            if (i > 0) {
                cout << " + ";
            }
            cout << coeffs[i] << "x^" << i;
        }
        cout << endl;
    }
};
int main() {
    vector<double> c = { 1, 2, 3 };
    Polynomial p(c);
    p.display();
    double x = 2.0;
    //double result = p(x);
    cout << "Result of evaluating the polynomial for x = " << x << ": " << result << endl;
    return 0;
}
```

Fig. 17 (In-Lab Task)

Post-Lab Activities:

Task 01: Convert 2D to 1D **[Estimated time 40 minutes / 30 marks]**

- Write a C++ program that defines a Matrix class representing a 2D matrix of integers
- Overload the cast operator to convert a Matrix object to a 1D vector of integers
- The program should then create a Matrix object, convert it to a vector, and output the result to the console

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's

Division of Pre-Lab marks:	**[20 marks]**
Task 01: Big Integer	[20 marks]
Division of In-Lab marks:	**[80 marks]**
Task 01: Students GPA	[20 marks]
Task 02: Cast Complex Numbers	[30 marks]
Task 03: Overloading Polynomials	[30 marks]
Division of Post-Lab marks:	**[30 marks]**
Task 01: Convert 2D to 1D	[30 marks]

References and Additional Material:

- **Unary Operator Overloading**
  https://www.tutorialspoint.com/cplusplus/unary_operators_overloading.htm
- **Dynamic Memory Management**
  https://www.programiz.com/cpp-programming/memory-management
- **Overloading Function Call Operator**
  https://www.ibm.com/docs/en/i/7.2?topic=only-overloading-function-calls-c

Lab Time Activity Simulation Log:

• Slot – 01 – 00:00 – 00:15:	Class Settlement
• Slot – 02 – 00:15 – 00:40:	In-Lab Task
• Slot – 03 – 00:40 – 01:20:	In-Lab Task
• Slot – 04 – 01:20 – 02:20:	In-Lab Task
• Slot – 05 – 02:20 – 02:45:	Evaluation of Lab Tasks
• Slot – 06 – 02:45 – 03:00:	Discussion on Post-Lab Task