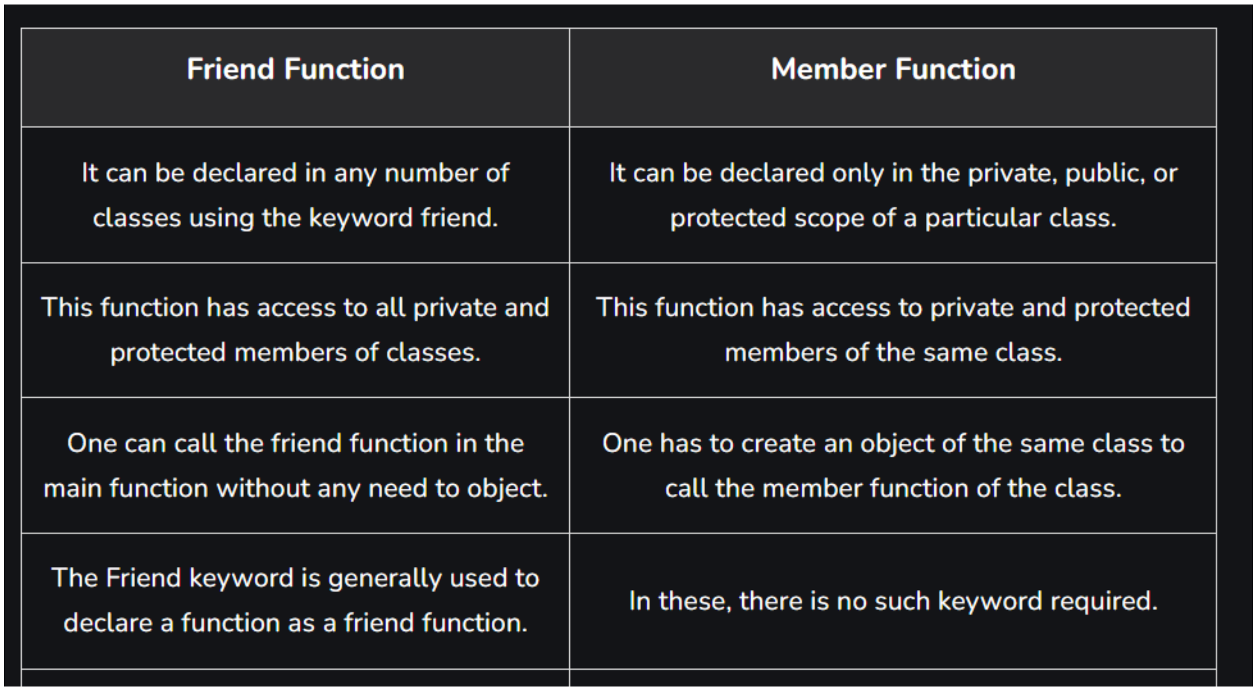
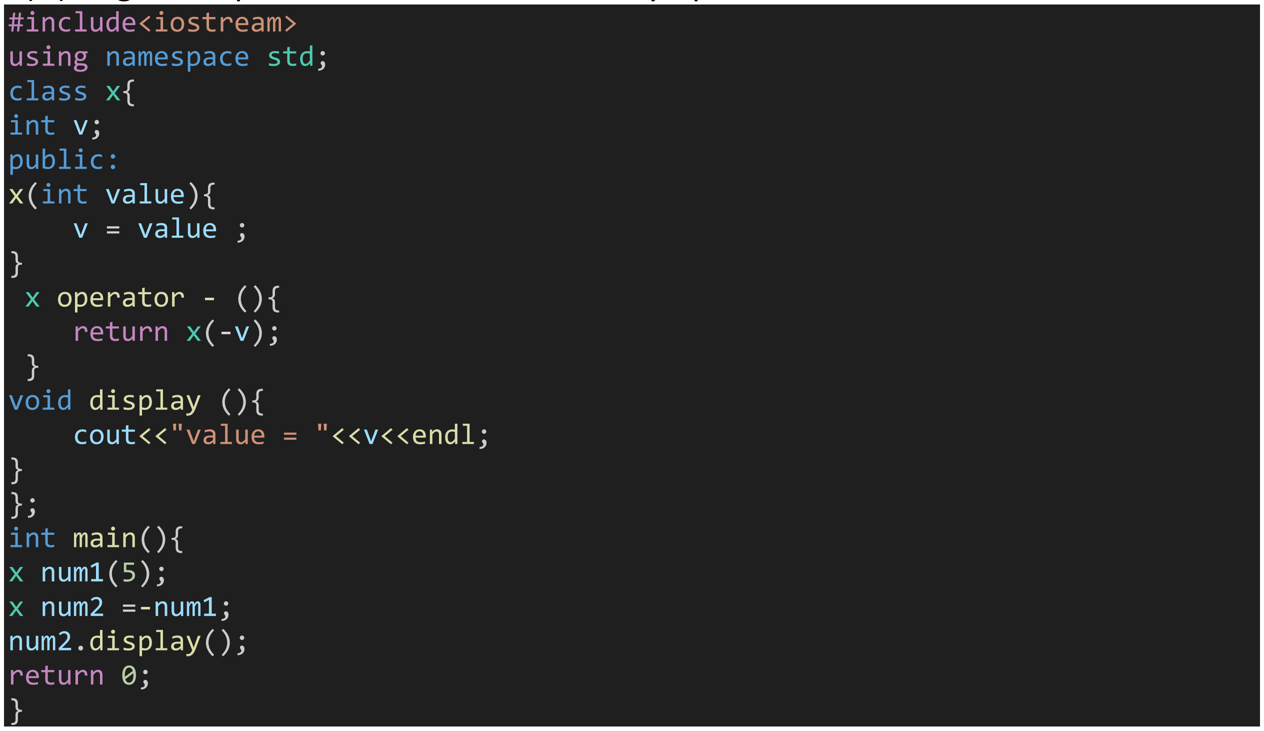
C++ previous question solving (Spring-2023)

1)a) To assign more than one operation on an same operator known as operator overloading.
Difference between member function and friend function:



Friend Function	Member Function
It can be declared in any number of classes using the keyword friend.	It can be declared only in the private, public, or protected scope of a particular class.
This function has access to all private and protected members of classes.	This function has access to private and protected members of the same class.
One can call the friend function in the main function without any need to object.	One has to create an object of the same class to call the member function of the class.
The Friend keyword is generally used to declare a function as a friend function.	In these, there is no such keyword required.

1)b) negative operator overloaded as unary operator:



```
#include<iostream>
using namespace std;
class x{
int v;
public:
x(int value){
    v = value ;
}
 x operator - (){
    return x(-v);
 }
void display (){
    cout<<"value = "<<v<<endl;
}
};
int main(){
x num1(5);
x num2 =-num1;
num2.display();
return 0;
}
```

Solved by Nazrana

negative operator overloaded as binary operator:

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

class x {
private:
    int v, w;

public:
    // Default constructor
    x() : v(0), w(0) {}

    // Parameterized constructor
    x(int r, int t) : v(r), w(t) {}

    void display() {
        cout << "v = " << v << endl;
        cout << "w = " << w << endl;
    }

    // Overload the binary negative operator (-) as a friend function
    friend x operator-(x &obj, x &obj1);
};

// Definition of the binary negative operator
x operator-(x &obj, x &obj1) {
    x tmp;
    tmp.v = obj.v - obj1.v;
    tmp.w = obj.w - obj1.w;
    return tmp;
}

int main() {
    x ob(100, 60), ob1(30, 40), ob2;
    ob2 = ob - ob1;
    ob2.display();
    return 0;
}
```

1)b)or) according to question the code has written down :

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

class x {
 int v;

public:
    x(int value) {
        v = value;
    }
friend x operator +(int y , x &ob) {
        return (y+ob.v);
    }

    void display() {
        cout << "value = " << v << endl;
    }
};

int main() {
    x num1(20); // Added a semicolon to end this line
    x num2 = 50+num1;
    num2.display();
    return 0;
}
```

1)c) writing a program to overload ++using member function:

```
#include <iostream>
using namespace std;
class x{
int t;
public:
x(int val){
   t=val;
}
x& operator++(){
   ++t;
   return *this;
}
void display(){
   cout<<"value ="<<t<<endl;}
};
int main(){
x num(9);
num.display();
++num;
num.display();
}
```

1)c)or) overloaded && operator :

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

C++ previous question solving (Spring-2023)

```
class x {
    bool t;

public:
    x(bool val) : t(val) {}

    bool operator&&(const x& j) const {
        return t && j.t;
    }

    void display() {
        cout << "value = " << t << endl;
    }
};

int main() {
    x num(true); // Initialize with a boolean value
    num.display();

    x other(true); // Initialize another x object with a boolean value
    other.display();

    bool result = num && other; // Using the overloaded && operator

    cout << "Logical AND result: " << result << endl; // Print the result

    return 0;
}
```

1)d) corrected code:

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

class A {
private:
    int a, b;

public:
    A(int i, int j) : a(i), b(j) {}
    A(){a=0,b=0;}
A operator+(int i) {
    A temp;
    temp.a = a + i;
    temp.b = b + i;
    return temp;
}
```

```
    void show() {
        cout << a << " " << b << endl;
    }
};

int main() {
    A ob1(10, 5), ob2;
    ob2 = ob1 + 10;
    ob2.show();

    return 0;
}
```

2)a) there are two derived classes, D1 and D2 both of which inherit from a base class B. However, the key difference between them is the access specifier used in the inheritance:
class D1: private B
In this case, D1 is inheriting from B as a private base class.
All public and protected members of B will become private members of D1. This means that these members will not be accessible outside of the class D1.The base class B itself is not accessible from outside of D1.This is known as "private inheritance,"class D2: public B: In this case, D1 inherits from B using the public access specifier. This means that the public and protected members of B remain public and protected members, respectively, in D1.
class D2: public B : In this case, D2 is inheriting from B as a public base class.All public and protected members of B will retain their access specifiers in D2. Public members will remain public, and protected members will remain protected.The base class B itself is also accessible from outside of D2.This is a traditional "public inheritance.
2)a)or)

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

class Base {
    public:
    int u;
public:
    Base(int value) {
        u = value;
      cout << "Base constructor with value: " << u <<endl;
    }
};

class Derived : public Base {
```

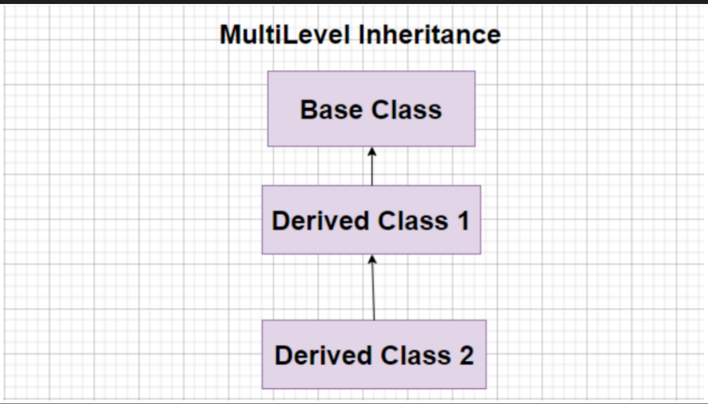
```
public:
int k;
    Derived(int de, int ba) : Base( ba) {
           k=de;

           cout << "Derived constructor with valueDerived: " << de<<endl;
    }
};

int main() {
    Derived d(10, 20); // This will call the Base constructor with
                      // valueBase = 20 and the Derived constructor with
valueDerived = 10.

     return 0;
}
```

2)b)



```
// C++ program to implement
// Multilevel Inheritance
#include <bits/stdc++.h>
using namespace std;

// single base class
class A {
public:
    int a;
    void get_A_data()
    {
        cout << "Enter value of a: ";
        cin >> a;
    }
};

// derived class from base class
class B : public A {
public:
    int b;
    void get_B_data()
    {
        cout << "Enter value of b: ";
        cin >> b;
    }
};
```

```
// derived from class derive1
class C : public B {
private:
    int c;

public:
    void get_C_data()
    {
        cout << "Enter value of c: ";
        cin >> c;
    }

    // function to print sum
    void sum()
    {
        int ans = a + b + c;
        cout << "sum: " << ans;
    }
};
int main()
{
    // object of sub class
    C obj;

    obj.get_A_data();
    obj.get_B_data();
    obj.get_C_data();
    obj.sum();
    return 0;
}
```

2)b)or) **virtual base class**: When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class.

```
Example of virtual base class:
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

class A
{
public:
    int i;
};
```

```
class B : virtual public A
{
public:
    int j;
};

class C : virtual public A
{
public:
    int k;
};

class D : public B, public C
{
public:
    int sum;
};

int main()
{
    D ob;
    ob.i = 10; // unambiguous since only one copy of i is inherited.
    ob.j = 20;
    ob.k = 30;
    ob.sum = ob.i + ob.j + ob.k;
    cout << "Value of i is: " << ob.i << "\n";
    cout << "Value of j is: " << ob.j << "\n";
    cout << "Value of k is: " << ob.k << "\n";
    cout << "Sum is: " << ob.sum << "\n";

    return 0;}
```

2)c) yes there are syntax errors. Here is the corrected code and output below:

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

class A {
public:
    void cheers() {
        cout << "Class A: Hip-hip-hooray" << endl;
    }
};
class B {
public:
    void cheers() {
        cout << "Class B: Hip-hip-hooray" << endl;
    }
};
```

```cpp
class C : public A, public B {
public:
};
int main() {
    C obc;
    obc.A::cheers();
    return 0;
}
//Output: Class A: Hip-hip-hooray
```

2)d)

```cpp
#include <iostream>
using namespace std;
// Base class
class Vehicle {
protected:
    int numWheels;
    double range;
public:
    Vehicle(int wheels, double r) : numWheels(wheels), range(r) {}
 void displayInfo() {
        cout << "Number of wheels: " << numWheels << endl;
        cout << "Range: " << range << " miles" << endl; }
};
// Derived class Car
class Car : public Vehicle {
private:
    int numPassengers;
public:
    Car(int wheels, double r, int passengers) : Vehicle(wheels, r),
numPassengers(passengers) {}
 void displayInfo() {
        Vehicle::displayInfo(); // Call the displayInfo function of the base
class
        cout << "Number of passengers: " << numPassengers << endl; }
};
// Derived class Truck
class Truck : public Vehicle {
private:
    double loadLimit;
public:
    Truck(int wheels, double r, double limit) : Vehicle(wheels, r),
loadLimit(limit) {}
void displayInfo() {
        Vehicle::displayInfo(); // Call the displayInfo function of the base
class
        cout << "Load limit: " << loadLimit << " tons" << endl;}};
```

C++ previous question solving (Spring-2023)

```
int main() {
    // Create objects of the derived classes
    Car myCar(4, 400, 5);
    Truck myTruck(6, 600, 3.5);
 // Display information for each object
    cout << "Car Information:\n";
    myCar.displayInfo();
    cout << "\nTruck Information:\n";
    myTruck.displayInfo();
 return 0;}
```

3)a)
```
#include <iostream>
#include <string>
using namespace std;
class Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Animal() : type("Animal") {}

    // declare virtual function
    virtual string getType() {
        return type;
    }};
class Dog : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Dog() : type("Dog") {}

    string getType() override {
        return type;   }
};
class Cat : public Animal {
   private:
    string type;
 public:
    // constructor to initialize type
    Cat() : type("Cat") {}
 string getType() override {
        return type; }
};
void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;}
```

```
int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();
print(animal1);
    print(dog1);
    print(cat1);
 return 0;}
```

3)b)

Early binding:

Early binding essentially refers to those events that can be known at compile time. Specifically, it refers to those function calls that can be resolved during compilation.

Late binding:

Late binding refers to events that must occur at run time. A late bound function call is one in
which the address of the function to be called is not known until the program runs. In C++,a virtual function is a late bound object.

Pros and Cons of Early binding:

The main advantage of early binding (and the reason that it is so widely used) is that it is veryefficient. Calls to functions bound at compile time are the fastest types of function calls. Themain disadvantage is lack of flexibility.

Pros and Cons of Late binding:

The main advantage of late binding is flexibility at run time. Its primary disadvantage is that
there is more overhead associated with a function call. This generally makes such calls slower
than those that occur with early binding. Because of the potential efficiency trade-offs, you
must decide when it is appropriate to use early binding and when to use late binding.

```
3)c) #include <iostream>
using namespace std;
class Test {
private:
    int m;
public:
    void getData() {
        cout << "Enter number: ";
        cin >> m; }
  void display() {
        cout << m; }};
int main() {
    Test T;
    T.getData();
    T.display();
 Test *p = new Test;
    p->getData();
    p->display();
delete p; // Don't forget to free the dynamically allocated memory
    return 0;}
```

4)a)
```
#include <iostream>
#include <string>
using namespace std;
template<typename T>
T Add(T a, T b) {
    return a + b;
}
int main() {
    // Adding two integers
    int intResult = Add(5, 10);
    cout << "Adding two integers: 5 and 10. Result: " << intResult << endl;
// Adding two floating-point numbers
    double doubleResult = Add(3.14, 2.71);
    cout << "Adding two floating-point numbers: 3.14 and 2.71. Result: " <<
doubleResult << endl;
 // Concatenating two strings
    string strResult = Add(string("Hello"), string("World"));
    cout << "Concatenating two strings: \"Hello\" and \"World\". Result: " <<
strResult << endl;
return 0;}
```

4)a)or)
```
#include <iostream>
#include <vector>
#include <numeric>  // For std::accumulate
using namespace std;
template <typename T>
double calculateAverage(const vector<T>& arr) {
    if (arr.empty()) {
        cerr << "Error: Array is empty. Cannot calculate average.\n";
        return 0.0;
    }
    return accumulate(arr.begin(), arr.end(), 0.0) / arr.size();
}
int main() {
    // Array of integers
    vector<int> intArray = {10, 20, 30, 40, 50};
    cout << "Average of integers: " << calculateAverage(intArray) << endl;

    // Array of floating-point numbers
    vector<double> doubleArray = {3.5, 4.7, 2.9, 6.1, 1.8};
    cout << "Average of doubles: " << calculateAverage(doubleArray) << endl;

    // Array of characters
    vector<char> charArray = {'A', 'B', 'C', 'D', 'E'};
    cout << "Average of characters: " << calculateAverage(charArray) << endl;
    return 0;
}
```

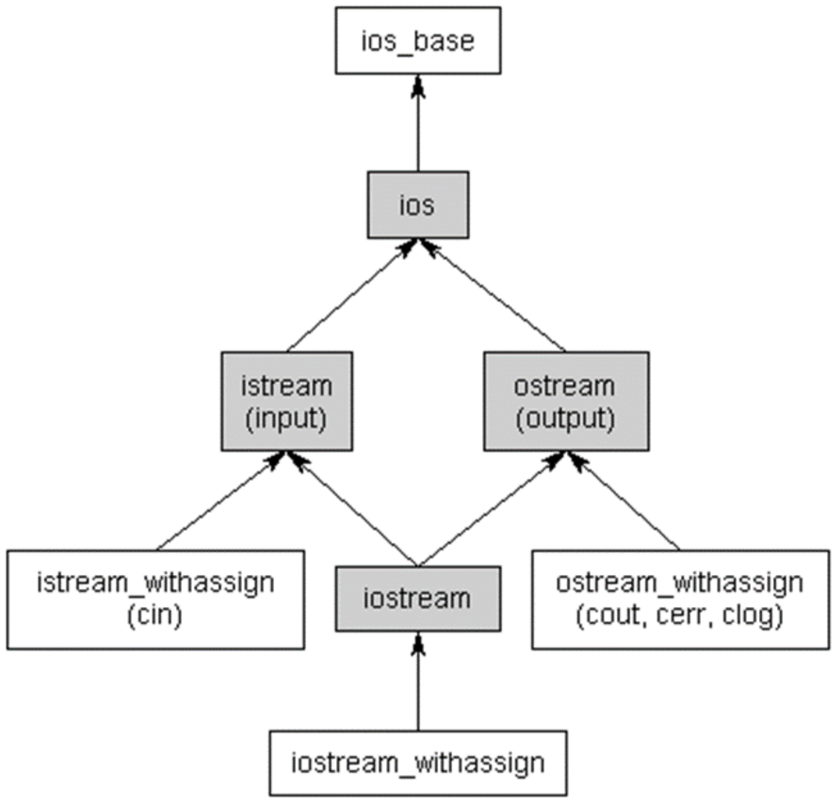
4)b)
```
#include <iostream>
#include <list>
using namespace std;
int main() {
    // Initialize a list with values (10, 20, 30, 40, 50)
    list<int> myList = {10, 20, 30, 40, 50};
 // Print the initial list
    cout << "Initial List: ";
    for (const auto& value : myList) {
        cout << value << " ";
    } cout << endl;
// Remove the element at index 2
    auto itRemove = next(myList.begin(), 2);
    myList.erase(itRemove);
 // Print the list after removing the element at index 2
    cout << "After removing element at index 2: ";
    for (const auto& value : myList) {
        cout << value << " ";
    } cout << endl;
// Insert the value 15 at index 1
    auto itInsert = next(myList.begin(), 1);
    myList.insert(itInsert, 15);
 // Print the list after inserting 15 at index 1
    cout << "After inserting 15 at index 1: ";
    for (const auto& value : myList) {
        cout << value << " ";
    }  cout << endl;
// Remove the first element from the list
    myList.pop_front();
    // Print the list after removing the first element
    cout << "After removing the first element: ";
    for (const auto& value : myList) {
     cout << value << " "; }
    cout << endl;
    return 0;
}
```

4)c) In programming, an exception is an abnormal or unexpected event that occurs during the execution of a program and disrupts its normal flow. Exceptions can be caused by various factors, such as invalid input, runtime errors, or external conditions that the program cannot handle. When an exception occurs, it typically leads to the termination of the normal program flow and jumps to a special section of code called an exception handler.

1. **Separation of Concerns:** Exception handling separates error-handling code, enhancing code cleanliness, readability, and maintainability.
2. **Robustness:** It makes programs more robust by handling unexpected situations, preventing abrupt crashes and improving overall stability.
3. **Code Clarity:** Promotes clear and readable code by isolating error-handling logic, keeping the main functionality focused.
4. **Error Reporting:** Provides a standardized way to report errors, replacing the need for error return codes and offering detailed error information.
5. **Program Flow Control:** Enables control over program flow during errors, allowing flexible handling at various call stack levels.
6. **Cleanup Actions:** Leverages RAII principles to ensure proper resource release even during exceptions, enhancing resource management.
7. **Consistent Error Handling:** Ensures consistent error-handling practices across code sections, eliminating the need for extensive error code checks.
8. **Extensibility:** Easily extends error-handling capabilities by defining custom exception classes, creating a hierarchy for diverse error scenarios.
9. **Debugging:** Simplifies debugging by halting program execution during exceptions, aiding in identifying and resolving issues efficiently.

5)a) Explanation of the hierarchy:
std::ios_base: The base class for input and output stream classes. It provides basic functionality and properties shared by both input and output streams.

C++ previous question solving (Spring-2023)



std::istream: The base class for input streams. It provides functions for reading input from various sources, including the console.
std::ostream: The base class for output streams. It provides functions for writing output to various destinations, including the console.
std::iostream: The base class for bidirectional streams. It inherits from both std::istream and std::ostream and is used for both input and output operations.
5)a)or)

1. ios functions returns value while manipulators does not.

2.we can not create own ios functions while we can create our own manipulators.

3.ios functions are single and not possible to be combined while manipulators are possible to be applied in chain.

4.ios function needs <iostream> while manipulators needs <iomanip>

5.ios functions are member functions while manipulators are non-member functions.
5)b)

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
```

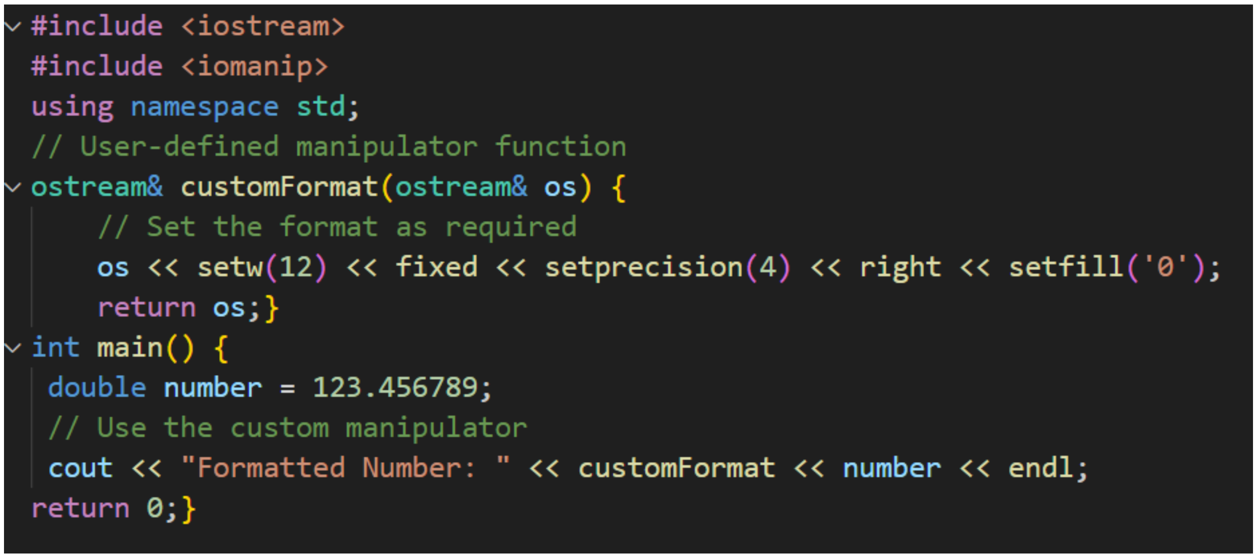
```cpp
using namespace std;
int main() {
    // Set up the random number generator
    srand(static_cast<unsigned>(time(nullptr)));
// Generate and write random numbers to "Numbers.txt"
    ofstream outFile("Numbers.txt");
    if (!outFile) {
        cerr<< "Error opening file for writing.\n";
        return 1;   }
 for (int i = 0; i < 50; ++i) {
        double randomNumber = static_cast<double>(rand()) / RAND_MAX;
        outFile << fixed << setprecision(6) << randomNumber << "\n";}
    outFile.close();
    // Read and display numbers from "Numbers.txt"
    ifstream inFile("Numbers.txt");
    if (!inFile) {
        cerr << "Error opening file for reading.\n";
        return 1;}
  double number;
    int count = 0;
cout << "Numbers from the file:\n";
    while (inFile >> number) {
        cout << fixed << setprecision(6) << number << "\n";
        ++count;}
inFile.close();
    cout << "Total numbers read: " << count << endl;
    return 0;}
```

5)c)

```cpp
 #include <iostream>
using namespace std;
 int main()
{   cout.setf(ios::showpos);
  cout.setf(ios::scientific);
  cout << 123 << " " << 123.23 << endl;

  cout.precision(2);                 // two digits after decimal point
  cout.width(10);                    // in a field of 10 characters
  cout << 123 << " ";
  cout.width(10);                    // set width to 10
  cout << 123.23 << endl;
  cout.fill('#');                    // fill using #
  cout.width(10);                    // in a field of 10 characters
  cout << 123 << " ";
  cout.width(10);                    // set width to 10
  cout << 123.23;
  return 0; }
```

5)c)or)



```
#include <iostream>
#include <iomanip>
using namespace std;
// User-defined manipulator function
ostream& customFormat(ostream& os) {
    // Set the format as required
    os << setw(12) << fixed << setprecision(4) << right << setfill('0');
    return os;}
int main() {
 double number = 123.456789;
 // Use the custom manipulator
 cout << "Formatted Number: " << customFormat << number << endl;
return 0;}
```

----------------------Finish--------------------------