FAST NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, PESHAWAR

DEPARTMENT OF COMPUTER SCIENCE

CL217 – OBJECT ORIENTED PROGRAMMING LAB

Friend Function and Friend class in C++

LAB MANUAL # 11

Instructor: Fariba Laiq

SEMESTER SPRING 2023

Friend Functions in C++

Friend functions of the class are granted permission to access private and protected members of the class in C++. They are defined globally outside the class scope. Friend functions are not member functions of the class. So, what exactly is the friend function?

A friend function in C++ is a function that is declared outside a class but is capable of accessing the private and protected members of the class. There could be situations in programming wherein we want two classes to share their members. These members may be data members, class functions. In such cases, we make the desired function, a friend to both these classes which will allow accessing private and protected data of members of the class.

Generally, non-member functions cannot access the private members of a particular class. Once declared as a friend function, the function is able to access the private and the protected members of these classes.

```
class class_name

{

  friend data_type function_name(arguments/s); //syntax of friend function.

};
```

Characteristics of Friend Function in C++

- The function is not in the 'scope' of the class to which it has been declared a friend.
- Friend functionality is not restricted to only one class
- Friend functions can be a member of a class or a function that is declared outside the scope of class.
- It cannot be invoked using the object as it is not in the scope of that class.
- We can invoke it like any normal function of the class.
- Friend functions have objects as arguments.
- It cannot access the member names directly and has to use dot membership operator and use an object name with the member name.
- We can declare it either in the 'public' or the 'private' section.

C++ program to demonstrate the working of friend function

```
#include <iostream>
using namespace std;
class Distance {
    private:
        int meter;
        // friend function
        friend int addFive(Distance);
    public:
        Distance() : meter(0) {}
};

// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output:

```
Distance: 5
```

Accessing private data members of a class from the member function of another class using friend function

```
#include <iostream>
using namespace std;
//forward declaration
class Distance;
class Add {
        public:
                int addFive(Distance d);
};
class Distance {
        private:
                int meter;
                // friend function
        public:
                int getDistance()
                {
                        return meter;
                }
                Distance(int meter) {
                        this->meter=meter;
                }
                friend int Add::addFive(Distance d);
};

int Add::addFive(Distance d) {
        //accessing private members from the friend function
        d.meter += 5;
        return d.meter;
}

int main() {
        Distance d(5);
        cout<<"Distance before: "<<d.getDistance()<<endl;
        Add a;
        cout<<"Distance now: "<<a.addFive(d);
        return 0;
}
```

Access members of two different classes using friend functions

```
#include <iostream>
using namespace std;
// forward declaration
class ClassB;
class ClassA {
            private:
                        int numA;
            public:
                        ClassA(int numA)
                        {
                                    this->numA=numA;
                        }
                        // friend function declaration
                        friend int add(ClassA, ClassB);
};
class ClassB {
            private:
                        int numB;
            public:
                        ClassB(int numB)
                        {
                                    this->numB=numb;
                        }

                        // friend function declaration
                        friend int add(ClassA, ClassB);
};
// access members of both classes
int add(ClassA objectA, ClassB objectB) {
            return (objectA.numA + objectB.numB);
}
int main() {
            ClassA objectA(4);
            ClassB object(6);
            cout << "Sum: " << add(objectA, objectB);
            return 0;
}
```

Output:

```
Sum: 10
```

Friend class

We can also use a friend Class in C++ using the friend keyword.

When a class is declared a friend class, all the member functions of the friend class become friend functions. If ClassB is a friend class of ClassA. So, ClassB has access to the members of classA, but not the other way around. Now in the following code, if we want the class Operations to access all the member functions of the Distance class, instead explicitly declaring all the functions of class Distance as friends, we just make the whole class as a friend.

Syntax:

```
friend class class_name;
```

It includes two classes, "Distance" and "Operations". The "Distance" class has a private member "meter" representing distance in meters and a public function "getDistance()" to return the distance. The "Operations" class has two friend functions, "addFive()" and "subFive()", which can access the private member of the "Distance" class. The main function creates an object of the "Distance" class and initializes it with a value of 5. It then creates an object of the "Operations" class and calls the "addFive()" and "subFive()" functions to increment and decrement the distance by 5, respectively.

Forward declaration is a technique in C++ that allows declaring the name of a class, function, or object before defining it. It informs the compiler that the name is a valid entity, and the definition will be provided later in the code.

```
#include <iostream>
using namespace std;
//forward declaration
class Distance;
class Operations {
        public:
                int addFive(Distance &d);
                int subFive(Distance &d);
};
class Distance {
        private:
                int meter;
                // friend function
        public:
                int getDistance()
                {
                        return meter;
                }
                Distance(int meter) {
                        this->meter=meter;
                }
                friend class Operations;
};

int Operations::addFive(Distance &d) {
        //accessing private members from the friend function
        d.meter += 5;
        return d.meter;
}

int Operations::subFive(Distance &d) {
        //accessing private members from the friend function
        d.meter -= 5;
        return d.meter;
}
int main() {
        Distance d(5);
        cout<<"Distance before: "<<d.getDistance()<<endl;
        Operations a;
        cout<<"Distance now by adding 5: "<<a.addFive(d)<<endl;
        cout<<"Distance now by subtracting 5: "<<a.subFive(d);
}
```

Output:

Distance before: 5

Distance now by adding 5: 10

Distance now by subtracting 5: 5

Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading. Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the data members of our objects.

Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Here,

returnType is the return type of the function.

operator is a keyword.

symbol is the operator we want to overload. Like: +, <, -, ++, etc.

arguments is the arguments passed to the function.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Using a Class Method:

In the following code, the prefix ++ operator is overloaded using the operator++() function, which increments the meter member variable by 1. The postfix ++ operator is overloaded using the operator++(int) function, which also increments the meter member variable by 1.

In the main() function, a Distance object d is created with an initial value of 5, and its value is printed using the display() function. The postfix ++ operator is then applied to d, which increments the meter member variable by 1 using the operator++(int) function and prints the message "Postfix increment" to the console. The updated value of d is printed using the display() function.

Next, the prefix ++ operator is applied to d, which increments the meter member variable by 1 using the operator++() function and prints the message "Prefix increment" to the console. The updated value of d is printed using the display() function. As the methods are defined inside the class so there is no need to pass the value to the operator function, as it will operate on the data members of the class directly.

```
#include <iostream>
using namespace std;
class Distance {
                int meter;
        public:
                Distance(int meter) {
                        this->meter=meter;
                }
                void display() {
                        cout<<"distance: "<<meter<<endl;
                }
                void operator ++()
                {
                        cout<<"Prefix increment"<<endl;
                        ++meter;
                }
                void operator ++(int)
                {
                        cout<<"Postfix increment"<<endl;
                        meter++;
                }
};
int main() {
        Distance d(5);
        d.display();
        d++;
        d.display();
        ++d;
        d.display();
}
```

Output:

```
distance: 5
Postfix increment
distance: 6
Prefix increment
distance: 7
```

Using a Friend Function:

The below code does exactly the same task as the previous one, the only difference is here the operator function is defined outside the class and declared as friend in the class so that it can directly access the private members of the class. In that case we have to pass one argument to the friend function.

```
#include <iostream>
using namespace std;
class Distance {
                    int meter;
          public:
                    Distance(int meter) {
                              this->meter=meter;
                    }
                    void display() {
                              cout<<"distance: "<<meter<<endl;
                    }
                    friend void operator ++(Distance &d);
                    friend void operator ++(Distance &d, int);
};
void operator ++(Distance &d) {
          cout<<"Prefix increment"<<endl;
          ++d.meter;
}

void operator ++(Distance &d, int) {
          cout<<"Postfix increment"<<endl;
          d.meter++;
}
int main() {
          Distance d(5);
          d.display();
          d++;
          d.display();
          ++d;
          d.display();
}
```

Output:

```
distance: 5
Postfix increment
distance: 6
Prefix increment
distance: 7
```

Operator Overloading in Binary Operators

Using a Class Method:

The operator+ function overloads the + operator for Distance objects. The function takes a reference to a Distance object d as input parameter and returns a new Distance object d3, which is the result of adding the meter member variable of the input Distance object and the meter member variable of the calling Distance object. The function also prints the message "Addition" to the console.

In the main() function, two Distance objects d1 and d2 are created with initial values of 5 and 3, respectively, and their values are printed using the display() function. The + operator is then applied to d1 and d2, which calls the operator+ function defined in the Distance class and returns a new Distance object d3, whose value is the sum of the meter member variables of d1 and d2. The value of d3 is printed using the display() function.

```
#include <iostream>
using namespace std;
class Distance {
                int meter;
        public:
                Distance(int meter) {
                        this->meter=meter;
                }
                void display() {
                        cout<<"distance: "<<meter<<endl;
                }
                Distance operator + (Distance &d) {
                        cout<<"Addition"<<endl;
                        int meter=this->meter+d.meter;
                        Distance d3(meter);
                        return d3;
                }
};
int main() {
        Distance d1(5);
        Distance d2(3);
        d1.display();
        d2.display();
        Distance d3=d1+d2;
        d3.display();

}
```

Output:

```
distance: 5
distance: 3
Addition
distance: 8
```

Using a Friend Function:

This code demonstrates the same logic as the previous ones, the only difference is that the operator function is defined outside the class and declared as a friend in the Distance class so that it can access the private data members of the class directly. As this function is defined outside the class so in that case of binary operator overloading, we have to pass both arguments to the function.

```
#include <iostream>
using namespace std;
class Distance {
        int meter;
    public:
        Distance(int meter) {
            this->meter=meter;
        }
        void display() {
            cout<<"distance: "<<meter<<endl;
        }
        friend Distance operator + (Distance&, Distance&);
};
Distance operator + (Distance &d1, Distance &d2) {
    cout<<"Addition"<<endl;
    int meter=d1.meter+d2.meter;
    Distance d(meter);
    return d;
}

int main() {
    Distance d1(5);
    Distance d2(3);
    d1.display();
    d2.display();
    Distance d3=d1+d2;
    d3.display();

}
```

Output:

```
distance: 5
distance: 3
Addition
distance: 8
```