Object-Oriented Programming

CL-1004

Lab 08

Friend Function, Friend Classes,
Operator Overloading

National University of Computer and Emerging Sciences–NUCES – Karachi

Contents

1. What is a Friend Function?

A **friend function** in C++ is a non-member function that has access to the private and protected members of a class. It is declared inside the class using the friend keyword. Unlike member functions, friend functions are **not part of the class** but can interact with its data.

Syntax:

```
class ClassName {
    private:
        // Private members
    public:
        // Public members

        // Declare a friend function
        friend ReturnType FunctionName(Parameters);
    };

    // Define the friend function (outside the class)
    ReturnType FunctionName(Parameters) {
        // Function implementation
    }
```

Essential Properties:

- Declared with **friend** inside the class.
- Can be defined anywhere in the program (like a normal function).
- Does not belong to the **class scope** (called without using the class object).

1.1. Why do we need Friend Functions?

Friend functions are useful when:

- A function needs to access **private/protected data of multiple classes**.
- Operator overloading requires direct access to class internals (e.g., operator<< for streams).
- Bridging two or more classes in a single operation.

Example:

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

// forward declaration of Humidity class | it allows the
Temperature class to reference the Humidity class before its full
definition.
class Humidity;

class Temperature {
private:
    float temp;
public:
    Temperature(float t) : temp(t) {}
    // declare friend function
    friend float calculateHeatIndex(Temperature t, Humidity h);
};

class Humidity {
private:
    float humidity; // humidity in %
public:
    Humidity(float h) : humidity(h) {}
    // declare friend function
    friend float calculateHeatIndex(Temperature t, Humidity h);
};

// friend function definition
float calculateHeatIndex(Temperature t, Humidity h) {

    float HI = -42.379 + 2.04901523 * t.temp
               + 10.14333127 * h.humidity
               - 0.22475541 * t.temp * h.humidity;
    return HI;
}

int main() {
    Temperature temp(32.0); // 32°C
    Humidity hum(70.0);     // 70% humidity
    float heatIndex = calculateHeatIndex(temp, hum);
    cout << "Heat Index is: " << heatIndex << "C" << endl;
    return 0;
}
```

Output	Heat Index is: 229.771C

1. **Forward Declaration:**

   Humidity is declared before **Temperature** to resolve dependency in the **friend** function declaration.

2. **Friend Declaration:**

   Both classes declare **calculateHeatIndex()** as a **friend**, granting it access to their **private** temp and humidity members.

3. **Heat Index Formula:**

   A simplified version of the National Weather Service's formula is used for demonstration.

2. What is a Friend Class?

A **friend class** in C++ is a class that has access to the **private** and **protected** members of another class. It is declared using the **friend** keyword inside the class granting access. Unlike inheritance, friend classes are **not hierarchical**—friendship is explicitly granted and does not imply a **parent-child relationship**.

Syntax:

```
class ClassName {
    private:
        // private members
    public:
        // public members

        // declare a friend class
        friend class FriendClassName;
    };

class FriendClassName {
        // friendClassName can now access private members of
ClassName
};
```

Essential Properties:

- Declared with **friend** class **ClassName**; in the host class.
- The friend class can access **all private/protected members** of the host class.
- Friendship is **one-way**: If ClassA declares ClassB as a **friend**, ClassB can access ClassA's **private** data, but ClassA cannot access ClassB's **private** data unless explicitly declared.

2.1. Why do we need Friend Classes?

Friend classes are useful when:

- A group of related functions in another class need frequent access to private data.
- Tight collaboration between two classes is required (e.g., a manager class overseeing multiple objects).
- Avoiding excessive getter/setter functions that expose internal data.

Example:

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

// forward declaration of Librarian class
class Librarian;

class Book {
private:
    string title;
    double price; // private price of the book

public:
    Book(string t, double p) : title(t), price(p) {}

    // declare Librarian as a friend class
    friend class Librarian;
};

class Librarian {
public:

    void displayBookDetails(const Book& book) {
        cout << "Book Title: " << book.title << endl;
        cout << "Book Price: $" << book.price << endl;
    }

    // function to apply a discount (modifying private members)
    void applyDiscount(Book& book, double discount) {
        book.price -= discount;
        cout << "Discounted Price: $" << book.price << endl;
    }
};

int main() {
    Book book("C++ Programming", 45.99);
    Librarian librarian;

    // librarian accesses and modifies private book details
```

```
    librarian.displayBookDetails(book);
    librarian.applyDiscount(book, 10.0); // apply $10 discount

    return 0;
}
```

Output	Book Title: C++ Programming Book Price: $45.99 Discounted Price: $35.99

1. **Book Class:**
    - Stores **private** data: title and price.
    - Declares **Librarian** as a **friend** class, allowing it to access title and price.
2. **Librarian Class:**
    - **displayBookDetails()**: Accesses and displays private book details.
    - **applyDiscount()**: Modifies the private price of the book.

3. Example: Friend Functions and Friend Classes

This example will simulate a Student-Teacher Grading System where:

- A Student class stores private student grades.
- A Teacher class (friend class) can modify and view student grades.
- A friend function calculates the average grade of a student.

Example:

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

// forward declaration
class Teacher;

class Student {
private:
    string name;
    double grades[3]; // private array to store 3 grades

public:
    Student(string n, double g1, double g2, double g3) : name(n) {
        grades[0] = g1;
        grades[1] = g2;
        grades[2] = g3;
```

```
    }

    // declare Teacher as a friend class
    friend class Teacher;

    // declare friend function to calculate average grade
    friend double calculateAverageGrade(const Student& student);
};

// friend function to calculate average grade
double calculateAverageGrade(const Student& student) {
    double sum = 0;
    for (int i = 0; i < 3; i++) {
        sum += student.grades[i];
    }
    return sum / 3;
}

class Teacher {
public:
    // modify a student's grade
    void updateGrade(Student& student, int index, double newGrade)
{
        if (index >= 0 && index < 3) {
            student.grades[index] = newGrade;
            cout << "Grade updated!" << endl;
        } else {
            cout << "Invalid grade index!" << endl;
        }
    }

    void viewGrades(const Student& student) {
        cout << "Grades for " << student.name << ":" << endl;
        for (int i = 0; i < 3; i++) {
            cout << "Grade " << i + 1 << ": " << student.grades[i]
<< endl;
        }
    }
};

int main() {

    Student student("Abdullah", 85.0, 90.0, 78.0);


    Teacher teacher;
```

```
    teacher.viewGrades(student);
    teacher.updateGrade(student, 1, 95.0); // update grade at
index 1
    teacher.viewGrades(student);

    // calculate average grade using friend function
    double average = calculateAverageGrade(student);
    cout << "Average Grade: " << average << endl;

    return 0;
}
```

Output

```
Grades for Abdullah:
Grade 1: 85
Grade 2: 90
Grade 3: 78

Grade updated!

Grades for Abdullah:
Grade 1: 85
Grade 2: 95
Grade 3: 78
Average Grade: 86
```

1. Student Class

- Stores **private** data: name and an array of grades.
- Declares Teacher as a **friend class** to allow access to its **private** members.
- Declares **calculateAverageGrade** as a **friend function** to allow access to its **private** members.

2. Teacher Class

- A **friend class** of Student.
- Has methods to:
    - **updateGrade**: Modify a student's grade.
    - **viewGrades**: Display a student's grades.

3. Friend Function

- **calculateAverageGrade** is a **friend function** that calculates the average grade of a student by accessing the **private** grades array.

4. Operator Overloading

Operator Overloading is a feature in C++ that allows you to redefine the behavior of operators (such as **+**, **-**, *****, **/**, **==**, **<<**, etc.) for user-defined types (e.g., classes and structures) and it comes under Compile-Time Polymorphism. Operator Overloading enables you to use operators with objects of your custom classes in a way that is intuitive and meaningful.

Essential Points:

1. **Purpose:**
    - To make user-defined types behave like built-in types.
    - To provide intuitive syntax for operations on objects.
2. **How It Works:**
    - You define a special function (called an **operator function**) that specifies what the operator should do when used with objects of your class.

4.1. Built In Overloads

Most operators are already overloaded for fundamental types.

Example:

1. In the case of the expression:

    a / b

    the operand type determines the machine code created by the compiler for the **division** operator. If both operands are integral types, an integral division is performed; in all other cases floating-point division occurs. Thus, different actions are performed depending on the operand types involved.

2. **<<**, which is used both as the stream insertion operator and as the bitwise left-shift operator.

4.2. Overloads For User Defined Types

Operators can be used with user-defined types as well. Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

Example:

The effect of **+** operator can be stipulated for the objects of a particular class.

4.3. Operator Function Syntax

To overload an operator, an appropriate **operator function is required**.

```
return_type operator op_symbol (arg_list){
    // function body - task defined
}
```

- **return_type** is the type of value returned by the specified operation.
- **op** is the operator being overloaded. (**+**,**-** ,etc)
- **op** is preceded by the keyword **operator**.

4.4. List of Operators that can be Overloaded

new	delete	new[]	delete[]	+	-	*	/	%
^	&	|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=	<<	>>
<<=	>>=	==	!=	<=	>=	&&	||	++
--	,	->*	->	()	[]			

4.5. List of Operators that can't be Overloaded

- **?:** (conditional)
- **.** (member selection)
- **.*** (member selection with pointer-to-member)
- **::** (scope resolution)
- **sizeof** (object size information)
- **typeid** (object type information)

Example:

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

class Rupee {
private:
    long data;

public:

    Rupee(int rupee = 0) : data(rupee) {}
```

```
    // unary minus operator.
    Rupee operator-() const {
        return Rupee(-data);
    }

    // addition operator.
    Rupee operator+(const Rupee& obj) const {
        return Rupee(data + obj.data);
    }

    // subtraction operator.
    Rupee operator-(const Rupee& obj) const {
        return Rupee(data - obj.data);
    }

    // addition assignment operator.
    Rupee& operator+=(const Rupee& obj) {
        data += obj.data;
        return *this;
    }

    // subtraction assignment operator.
    Rupee& operator-=(const Rupee& obj) {
        data -= obj.data;
        return *this;
    }

    friend ostream& operator<<(ostream& os, const Rupee& e);
};

ostream& operator<<(ostream& os, const Rupee& e) {
    os << e.data;
    return os;
}

int main() {
    Rupee wholesale(20), retail;
    retail = wholesale;
    cout << "Wholesale price: " << wholesale << endl;
    cout << "Retail price: " << retail << endl;

    Rupee discount(2);
    retail -= discount;
    cout << "Retail price including discount: " << retail << endl;

    wholesale = Rupee(34);
    cout << "New wholesale price: " << wholesale << endl;
```

```
    retail = wholesale + 10;
    cout << "New retail price: " << retail << endl;

    Rupee profit(retail - wholesale);
    cout << "The profit: " << profit << endl;

    profit = -profit;
    cout << "The profit after unary minus: " << profit << endl;

    return 0;
}
```

Output

```
Wholesale price: 20
Retail price: 20
Retail price including discount: 18
New wholesale price: 34
New retail price: 44
The profit: 10
The profit after unary minus: -10
```