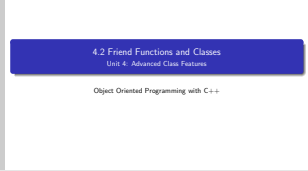
4.2 Friend Functions and Classes

Unit 4: Advanced Class Features

Object Oriented Programming with C++



Unit 4: Advanced Class Features

- Lecture 34: 4.2 Friend Functions and Classes
- Course: Object Oriented Programming with C++ (DI05011021)
- Topics: Friend Functions, Friend Classes, Encapsulation Considerations

Unit 4: Advanced Class Features
- Lecture 34: 4.2 Friend Functions and Classes
- Course: Object Oriented Programming with C++ (DI05011021)
- Topics: Friend Functions, Friend Classes, Encapsulation Considerations

Welcome back everyone. Today we are diving into lecture 34, which is a crucial part of Unit 4 on Advanced Class Features. We will be discussing Friend Functions and Friend Classes in C++. This topic addresses specific situations where the strict rules of encapsulation need to be carefully bypassed.

Recap: Strict Encapsulation

- Encapsulation binds data and functions together into a single unit (class).
- Private and protected members are hidden from the outside world.
- Only member functions of the class can access its private data.
- Non-member functions (global functions) have NO access to private members.
- Question: What if a global function absolutely needs access to private data of a class?

Recap: Strict Encapsulation
- Encapsulation binds data and functions together into a single unit (class).
- Private and protected members are hidden from the outside world.
- Only member functions of the class can access its private data.
- Non-member functions (global functions) have NO access to private members.
- Question: What if a global function absolutely needs access to private data of a class?

Let's start with a quick recap. We know that encapsulation is a fundamental pillar of OOP. It keeps our private data safe. Under normal circumstances, if a function is not a member of a class, the compiler will throw an error if it tries to access private variables. But in software design, rigid rules sometimes create complex workarounds. What if we genuinely have a scenario where an external function needs direct access to private members for efficiency or structural reasons?

Introduction to Friend Functions

- A 'friend' function is a special privilege granted by a class to a non-member function.
- It is NOT a member of the class.
- It CAN access private and protected members of the class in which it is declared as a friend.
- It bridges the gap between different classes or between a class and a global function.

Introduction to Friend Functions
- A 'friend' function is a special privilege granted by a class to a non-member function.
- It is NOT a member of the class.
- It CAN access private and protected members of the class in which it is declared as a friend.
- It bridges the gap between different classes or between a class and a global function.

This brings us to 'Friend Functions'. In C++, a class can grant 'friendship' to a specific external function. By doing so, the class explicitly allows that function to access its private and protected members. It is very important to understand that a friend function is NOT a member function. It doesn't belong to the class; it is just a trusted outsider.

Characteristics of Friend Functions

- Not in the scope of the class: Cannot be called using an object and the dot operator (e.g., obj.friendFunc() is invalid).
- Invoked like a normal global function.
- Cannot access member variables directly: Must use an object name and dot operator inside the function (e.g., obj.privateVar).
- Can be declared in the private or public section of the class with no difference in meaning.
- Usually has objects as arguments.

Characteristics of Friend Functions
- Not in the scope of the class: Cannot be called using an object and the dot operator (e.g., obj.friendFunc() is invalid).
- Invoked like a normal global function.
- Cannot access member variables directly: Must use an object name and dot operator inside the function (e.g., obj.privateVar).
- Can be declared in the private or public section of the class with no difference in meaning.
- Usually has objects as arguments.

Let's look at some key characteristics. Because it's not a member function, it doesn't have a 'this' pointer. You can't call it like 'object_name.function_name()'. You call it just like a regular C function. Since it doesn't have a 'this' pointer, if it wants to access a class's data, you must pass an object of that class to the friend function as an argument. Then, inside the function, you use that object to access the private data.

Declaring a Friend Function

- Use the keyword 'friend' inside the class definition.
- Syntax:
- class ClassName {
- // ... private members ...
- public:
- friend returnType functionName(arguments);
- };

Declaring a Friend Function
- Use the keyword 'friend' inside the class definition.
- Syntax:
- class ClassName {
- // ... private members ...
- public:
- friend returnType functionName(arguments);
- };

How do we make a function a friend? We do this by declaring the function prototype inside the class definition, preceded by the keyword 'friend'. The class itself must explicitly declare who its friends are. A function cannot force its way into a class and declare itself a friend. Friendship is granted, not taken.

Example: Basic Friend Function

```
class Distance {
private:
int meters;
public:
Distance() : meters(0) {}
// Friend declaration
friend void displayDistance(Distance d);
};

// Implementation (No 'friend' keyword, no scope resolution)
void displayDistance(Distance d) {
// Accessing private member 'meters' directly
cout ¡¡ "Distance: " ¡¡ d.meters ¡¡ " meters" ¡¡ endl;
}
```


Example: Basic Friend Function
class Distance {
private:
int meters;
public:
Distance() : meters(0) {}
// Friend declaration
friend void displayDistance(Distance d);
};

// Implementation (No 'friend' keyword, no scope resolution)
void displayDistance(Distance d) {
// Accessing private member 'meters' directly
cout ¡¡ "Distance: " ¡¡ d.meters ¡¡ " meters" ¡¡ endl;
}

Here is a simple code example. We have a 'Distance' class with a private variable 'meters'. We declare 'displayDistance' as a friend. Notice the implementation of 'displayDistance' at the bottom. It doesn't use the class name or the scope resolution operator because it's not a member function. Yet, inside the function, it can write 'd.meters' even though 'meters' is private.

Why Use Friend Functions?

- Operator Overloading: Especially for operators where the left-hand operand is not an object of the class (e.g., overriding the ¡¡ operator for cout).
- Bridging Multiple Classes: When a function needs to access private data of two entirely different classes simultaneously.
- Increasing Execution Speed: In specific real-time or high-performance systems where getter/setter overhead is undesirable (though rarely the sole reason in modern compilers).

Why Use Friend Functions?
- Operator Overloading: Especially for operators where the left-hand operand is not an object of the class (e.g., overriding the ¡¡ operator for cout).
- Bridging Multiple Classes: When a function needs to access private data of two entirely different classes simultaneously.
- Increasing Execution Speed: In specific real-time or high-performance systems where getter/setter overhead is undesirable (though rarely the sole reason in modern compilers).

You might wonder why we need this. The most common use case is when overloading the insertion (¡¡) and extraction (¿¿) operators for I/O streams. Because the left operand is a stream object (like cout) and not our class object, a member function won't work well. Another major use case is when a single function needs to act as a bridge between two different classes, comparing or calculating using private data from both.

Bridging Two Classes (Part 1)

```
// Forward declaration
class ClassB;

class ClassA {
private:
int valA;
public:
ClassA(int v) : valA(v) {}
// Friend function declaration
friend void compareValues(ClassA a, ClassB b);
};
```


Bridging Two Classes (Part 1)
// Forward declaration
class ClassB;
class ClassA {
private:
int valA;
public:
ClassA(int v) : valA(v) {}
// Friend function declaration
friend void compareValues(ClassA a, ClassB b);
};

Let's look at bridging two classes. We have ClassA and ClassB. We want a function to compare the private values inside both. Notice the forward declaration of ClassB at the top. The compiler needs to know that ClassB exists when it parses the friend declaration in ClassA. ClassA declares 'compareValues' as its friend.

Bridging Two Classes (Part 2)

```
class ClassB {
private:
int valB;
public:
ClassB(int v) : valB(v) {}
// The SAME function declared as friend here too
friend void compareValues(ClassA a, ClassB b);
};

void compareValues(ClassA a, ClassB b) {
if (a.valA ¿ b.valB) cout ¡¡ "A is greater";
else cout ¡¡ "B is greater (or equal)";
}
```


Bridging Two Classes (Part 2)
class ClassB {
private:
int valB;
public:
ClassB(int v) : valB(v) {}
// The SAME function declared as friend here too
friend void compareValues(ClassA a, ClassB b);
};

void compareValues(ClassA a, ClassB b) {
if (a.valA ¿ b.valB) cout ¡¡ "A is greater";
else cout ¡¡ "B is greater (or equal)";
}

Now we define ClassB. ClassB also declares the EXACT SAME function, 'compareValues', as its friend. Because the function is a friend to both classes, its implementation at the bottom can seamlessly access 'a.valA' and 'b.valB'. Both classes trusted this single external function.

Introduction to Friend Classes

- Just as functions can be friends, an entire class can be a friend of another class.
- If Class B is a friend of Class A, then ALL member functions of Class B can access private and protected members of Class A.
- It acts as a blanket permission.

Introduction to Friend Classes
- Just as functions can be friends, an entire class can be a friend of another class.
- If Class B is a friend of Class A, then ALL member functions of Class B can access private and protected members of Class A.
- It acts as a blanket permission.

What if we need many functions in another class to access our private data? Declaring twenty friend functions is messy. In C++, you can make an entire class a friend. If Class A declares Class B as a friend, then every single member function inside Class B is allowed to touch Class A's private parts. This is a very powerful feature.

Declaring a Friend Class

- Syntax:
- class Node {
- private:
- int data;
- Node* next;
- // Declaring LinkedList as a friend class
- friend class LinkedList;
- };
-
- class LinkedList {
- // All methods here can access Node::data and Node::next
- };

Declaring a Friend Class
Syntax:
class Node {
private:
int data;
Node* next;
// Declaring LinkedList as a friend class
friend class LinkedList;
};
class LinkedList {
// All methods here can access Node::data and Node::next
};

The syntax is simple. Inside the class granting friendship (like the Node class here), you use the keyword 'friend' followed by 'class' and the class name (LinkedList). A classic example is data structures. A Node class might hold data privately, but it grants the LinkedList class full access to manage those nodes.

Properties of Friendship in C++

- 1. Friendship is NOT mutual (Not symmetric):
- - If Class A is a friend of Class B, Class B is NOT automatically a friend of Class A.
- 2. Friendship is NOT inherited:
- - A derived class does not inherit the friends of its base class.
- 3. Friendship is NOT transitive:
- - If A is a friend of B, and B is a friend of C, A is NOT a friend of C.

Properties of Friendship in C++
- 1. Friendship is NOT mutual (Not symmetric):
- - If Class A is a friend of Class B, Class B is NOT automatically a friend of Class A.
- 2. Friendship is NOT inherited:
- - A derived class does not inherit the friends of its base class.
- 3. Friendship is NOT transitive:
- - If A is a friend of B, and B is a friend of C, A is NOT a friend of C.

It's vital to understand the rules governing friendship. First, it is not mutual. Just because my neighbor has keys to my house doesn't mean I have keys to theirs. Class A granting access to B doesn't mean A gets access to B. Second, friendship doesn't pass down through inheritance. A child class doesn't automatically trust its parent's friends. Finally, it's not transitive. The friend of my friend is not automatically my friend.

Friend Function vs. Member Function

- Member Function:
- - Declared and defined within class scope.
- - Called using object (obj.func()).
- - Has implicit 'this' pointer.
- - Naturally has access to private members.
-
- Friend Function:
- - Declared with 'friend' inside class, defined outside.
- - Called like a normal function (func(obj)).
- - No 'this' pointer; must pass object explicitly.
- - Has special permission to access private members.

Friend Function vs. Member Function
Member Function:
- Declared and defined within class scope.
- Called using object (obj.func()).
- Has implicit 'this' pointer.
- Naturally has access to private members.

Friend Function:
- Declared with 'friend' inside class, defined outside.
- Called like a normal function (func(obj)).
- No 'this' pointer; must pass object explicitly.
- Has special permission to access private members.

To summarize the structural differences: a member function belongs to the object. A friend function does not. This fundamental difference dictates how they are called and how they access data. Remember the 'this' pointer—member functions have it, friend functions do not.

The Debate: Does Friendship Break Encapsulation?

- Argument against:
- - It allows external entities to bypass data hiding, potentially making code harder to maintain and debug.
- Argument for:
- - It is EXPLICIT. A class must explicitly grant access.
- - Without it, programmers might make data public just to allow external access, which is much worse.
- - It provides a controlled, documented loophole rather than a complete breakdown of security.

The Debate: Does Friendship Break Encapsulation?
- Argument against:
- - It allows external entities to bypass data hiding, potentially making code harder to maintain and debug.
- Argument for:
- - It is EXPLICIT. A class must explicitly grant access.
- - Without it, programmers might make data public just to allow external access, which is much worse.
- - It provides a controlled, documented loophole rather than a complete breakdown of security.

There is a classic debate in OOP. Does the 'friend' keyword break encapsulation? Purists sometimes argue it does because data is no longer strictly hidden. However, Bjarne Stroustrup, the creator of C++, argues the opposite. Because friendship must be granted explicitly from within the class, it is a controlled opening. It's better to give one specific function the key than to leave the front door wide open by making the data public.

Best Practices for Using Friends

- 1. Use sparingly: Rely on public getters/setters when possible.
- 2. Keep it targeted: Prefer friend functions over friend classes if only one function needs access.
- 3. Use for tightly coupled classes: E.g., Matrix and Vector math classes, or structural nodes and container classes.
- 4. Standard idioms: Use for overloading operator¡¡ and operator¿¿.

Best Practices for Using Friends
- 1. Use sparingly: Rely on public getters/setters when possible.
- 2. Keep it targeted: Prefer friend functions over friend classes if only one function needs access.
- 3. Use for tightly coupled classes: E.g., Matrix and Vector math classes, or structural nodes and container classes.
- 4. Standard idioms: Use for overloading operator¡¡ and operator¿¿.

As for best practices, use friends sparingly. Don't use them just to save a few keystrokes writing getters and setters. If only one function needs access, don't make the whole class a friend. Use them when classes are naturally intertwined conceptually, like Matrix and Vector math, or linked lists and nodes.

Summary & Q&A

- Friend functions and classes can access private/protected members.
- They are declared using the 'friend' keyword inside the class.
- Friend functions do not have a 'this' pointer.
- Friendship is one-way, non-transitive, and non-inheritable.
- They provide controlled flexibility without completely abandoning encapsulation.
- Questions?

Summary & Q&A
- Friend functions and classes can access private/protected members.
- They are declared using the 'friend' keyword inside the class.
- Friend functions do not have a 'this' pointer.
- Friendship is one-way, non-transitive, and non-inheritable.
- They provide controlled flexibility without completely abandoning encapsulation.
- Questions?

To wrap up, we've learned how friend functions and classes work, the syntax to declare them, and the strict rules governing their use. We also discussed how they fit into the broader philosophy of encapsulation in C++. Are there any questions on how friendship works or when you should use it?