2.7 Friends

Sam found adding operator overloading functions to his `Date` class to be tedious. Because these functions were not members of his class, he was restricted to using public methods. This meant he needed to use the publicly-exposed properties rather than the privately-available data which would be more convenient.

Objectives

By the end of this chapter, you will be able to:

- Define and explain the uses for the `friend` modifier.
- Use the `friend` modifier to make a non-member function a `friend`.
- Explain the pros and cons of using `friend`s.

Prerequisites

Before reading this chapter, please make sure you are able to:

- List and define the rules of encapsulation (Chapter 2.0)
- Create a class matching a UML definition (Chapter 2.2 – 2.5)
- Overload common operators using non-member functions (Chapter 2.6)

What are friends and why you should care

Friends are special functions in C++ that have access to the private member variables and private methods of a class. Many would say that this violates the rules of encapsulation: giving non-members access to privates. After all, only member functions should have access to private member variables and functions!

The advantage of friends is that it is often desirable to allow some functions to operate directly on the data of a class without having to work with public properties. This is especially true when a non-trivial computation needs to happen that translate private data to public properties. It may be undesirable to expose these properties to the client. With friends, you don't. The class enumerates all the functions that have access to the privates thereby avoiding the unchecked public access to privates. In other words, the class gets to indicate which functions act like methods.

Syntax of friends

As you may recall, function prototypes typically go in header files. Member functions are also in the form of prototypes in the header file except they go inside the class definition. This is an important detail; classes have complete control over which functions are their friends.

To illustrate this point, consider the following class definition of our `Complex` class from last chapter:

```
class Complex
{
... code removed for brevity ...
};

// this inline function is defined outside the class definition
inline std::ostream & operator << (std::ostream & out, const Complex & rhs)
{
   out << rhs.getReal();                              // access the public methods
   if (rhs.getImaginary() != 0.0)                     //    getReal() and getImaginary()
      out << " + " << rhs.getImaginary() << "i";      //    just like all other non-member
   return out;                                        //    functions
}
```

In this example, observe how the inline function "`operator <<`" is defined outside the `Complex` class and how it only has access to the public methods `getReal()` and `getImaginary()`. To make the same function a `friend`, the prototype would be moved inside the `Complex` class definition and the `friend` keyword would be added:

This designates the function as a friend.
This will be defined inside the class definition.

```
class Comple
{
... code removed for brevity ...
   public:
      inline friend std::ostream & operator << (std::ostream & out,
                                                const Complex & rhs)
      {
         out << rhs.r;                                 // now we have access to the private
         if (rhs.i != 0.0)                             //    member variables 'r' and 'i'
            out << " + " << rhs.i << "i";              //    without having to go through
         return out;                                   //    the public methods
      }
... code removed for brevity ...
};
```

From this example, we can observe the three things that are unique about `friends`:

1. The `friend` keyword is added before the return type to the function prototype or inline function definition. This identifies it as a `friend`.
2. The function prototype or inline function definition resides inside the class definition
3. The function has access to private member variables and private methods

Note that if a friend function is defined inside a class definition, the `inline` keyword is optional. The compiler knowns the function should be inline simply because the function is defined inside a class definition.

Friends and operator overloading

Friends and operator overloading often occur hand-in-hand because operator overloading functions often feel like member functions. All the operator overloading functions taking an object as either the left-hand-side or the right-hand-side are tightly connected to the class they serve. Consider, for example, the non-friend version of the equivalence operator (==) defined for the `Card` class from previous chapters:

```
// Equivalence without using friends
bool operator == (const Card & lhs, const Card & rhs)
{
   // only same if both suit and rank are the same
   return lhs.getSuit() == rhs.getSuit() &&
          lhs.getRank() == rhs.getRank();
}
```

This is quite inconvenient. After all, we know that the private data implementation of the `Card` is a single `unsigned char` representing a value from 0 – 51. It would be much easier to just compare that value directly. Unfortunately, in order to do so, it is necessary to write another method exposing `data`:

```
class Card:
{
   public:
... code removed for brevity ...
   // functions that are public but really should be private
   int  getValue()    const { return value;            }
   void setValue(int value) { this->value = value;     }
... code removed for brevity ...
};
```

This would allow for a more convenient definition of the equivalence operator:

```
// Equivalence using a private method getValue()
bool operator == (const Card & lhs, const Card & rhs)
{
   // only same if the values are the same
   return lhs.getValue() == rhs.getValue();
}
```

If, on the other hand, we make the equivalence operator a `friend`, we can avoid defining `getValue()`:

```
class Card:
{
   public:
... code removed for brevity ...
      // there is now no need for getValue()
      inline friend bool operator == (const Card & lhs, const Card & rhs)
      {
         // only same if the values are the same
         return lhs.value == rhs.value;
      }
... code removed for brevity ...
};
```

Are friends bad?

Some object-oriented purists believe that friends represent a violation of encapsulation. This, strictly speaking, is true. Only member functions should have access to private member functions and member variables. Every function given access to privates represents more code that is made aware of the private and internal workings of a class. The rules of encapsulation stipulate that no one aside from members of the class should know or care how the internal workings of the class operate.

There are two counter-arguments to this point. First, only the class itself can designate whether a given function is a friend. In other words, a function cannot designate itself to be a friend of a class willy-nilly. First approval must be granted! Therefore friends do not represent wide-spread and wanton expansion of scope. The number of functions (member functions or friends) having access to privates are strictly controlled by the class.

The second counter-argument is significantly more powerful than the first. There are some operations that can only be effectively done with direct access to member variables. If a non-member function is to perform these operations, then it either must be made a `friend` or a special method must be written to accommodate it. The question is: which is a greater violation of encapsulation? Is it worse to give a handful of functions `friend` status or is it worse to make a data-centric (as opposed to a property-centric) method publically accessible? For example, our `Card` class needed to implement a `getValue()` method to efficiently implement the equivalence operator. This method exposed the internal workings of the `Card` class: that Spades came before Hearts in the suit order. The client should never be exposed to this implementation detail!

Because operator-overloading functions are closely tied to a class and because they typically need to have direct access to a class's private member variables, it is frequently best to make them friends.

Example 2.7 – Card

Demo

This example will demonstrate how to implement all the common non-member operators using friends. It would be helpful to compare this with the same code from last chapter to see the differences.

Problem

Modify the `Card` class to implement the common operators without having to explose the private data implementation. In other words, remove the `getValue()` and `setValue()` methods that were necessary before. .

Solution

The class definition is:

```
class Card
{
   public:
```

... code removed for brevity ...

```
      // insertion and extraction operators
      friend std::ostream & operator << (std::ostream & out, const Card & card);
      friend std::istream & operator >> (std::istream & in,        Card & card);

      // increment and decrement ... only changing rank
      friend Card & operator ++ (Card & lhs);                 // prefix
      friend Card & operator -- (Card & lhs);                 // prefix
      friend Card operator ++ (Card & lhs, int postfix);      // postfix
      friend Card operator -- (Card & lhs, int postfix);      // postfix

      // change a card by adding or subtracting one
      friend Card operator + (const Card & lhs, const int input);
      friend Card operator + (const int input, const Card & lhs);
      friend Card operator - (const Card & lhs, const int input);
      friend Card & operator += (Card & lhs, const int input);
      friend Card & operator -= (Card & lhs, const int input);

      // Relative comparison... only comparing rank
      friend bool operator >= (const Card & lhs, const Card & rhs);
      friend bool operator >  (const Card & lhs, const Card & rhs);
      friend bool operator <= (const Card & lhs, const Card & rhs);
      friend bool operator <  (const Card & lhs, const Card & rhs);

      // Absolute comparison... comparing both rank and suit
      friend bool operator == (const Card & lhs, const Card & rhs);
      friend bool operator != (const Card & lhs, const Card & rhs);
};
```

Observe how all the prototypes are in the class definition.

Challenge

As a challenge, see if you can make all the operators `inline` whereas currently they are not.

See Also

The complete solution is available at 2-7-card.html or:

```
/home/cs165/examples/2-7-card/
```

Example 2.7 – Complex

Demo

This is a second example on how to implement all the common operators with friends. Unlike "Example 2.7 – Card," all the overloaded functions are done as inlines.

Problem

Modify the `Complex` class to implement the common operators without having to expose the private data implementation.

Solution

The class definition is:

```
class Complex
{
   public:
```

... code removed for brevity ...

```
      inline friend std::ostream & operator << (std::ostream & out,
                                                const Complex & rhs)
      {
         out << rhs.r;
         if (rhs.i != 0.0)                       // only display the imaginary
            out << " + " << rhs.i << "i";        //    component if non-zero
         return out;                             // return "out"
      }
      inline friend std::istream & operator >> (std::istream & in,
                                                Complex & rhs)
      {
         in >> rhs.r >> rhs.i;      // input directly into the member varibles
         return in;                 // do not forget to return "in"
      }

      inline friend Complex operator + (const Complex & lhs,
                                        const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         return Complex(lhs.r + rhs.r, lhs.i + rhs.i);
      }

      inline friend Complex & operator += (Complex & lhs, const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         lhs.r += rhs.r;
         lhs.i += rhs.i;
         return lhs;
      }

      inline friend bool operator == (const Complex & lhs, const Complex & rhs)
      {
         return (lhs.r == rhs.r && lhs.i == rhs.i);
      }
```

... code removed for brevity ...

```
};
```

Observe how all the prototypes are in the class definition.

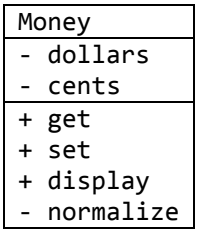
See Also

The complete solution is available at 2-7-complex.html or:

```
/home/cs165/examples/2-7-complex/
```

Review 1

Given the following UML class diagram:



Which operators should be overloaded?

Please see page 166 *for a hint.*

Problem 2

Given the following class definition:

```
class Money
{
   public:
      Money() : dollars(0), cents(0) { }
      Money(const Money & rhs) : dollars(rhs.dollars), cents(rhs.cents) {}
      double get() const { return (double)dollars + (double)cents / 100.0 }
      void set(double money);
   private:
      int  dollars;
      int  cents;
      void normalize();
};
```

Write the insertion and extraction operators using friends.

Please see page 185 *for a hint.*

Problem 3-6

Given the following class definition:

```
class Money
{
   public:
      Money() : dollars(0), cents(0) { }
      Money(const Money & rhs) : dollars(rhs.dollars), cents(rhs.cents) {}
      double get() const { return (double)dollars + (double)cents / 100.0 }
      void set(double money);
   private:
      int  dollars;
      int  cents;
      void normalize();
};
```

3. Write the addition (+) operator using friends.

4. Write the add-onto (+=) operator using friends.

5. Write the both the increment (++) operators (prefix and postfix) using friends.

6. Write the the equivalence (==) and greater-than (>) operators using friends.

Please see page 185 *for a hint.*