Chapter 15: Friends

In all examples discussed up to now, we've seen that `private` members are only accessible by the members of their class. This is *good*, as it enforces encapsulation and data hiding. By encapsulating functionality within a class we prevent that a class exposes multiple responsibilities; by hiding data we promote a class's data integrity and we prevent that other parts of the software become implementation dependent on the data that belong to a class.

In this (very) short chapter we introduce the `friend` keyword and the principles that underly its use. The bottom line being that by using the `friend` keyword functions are granted access to a class's `private` members. Even so, this does not imply that the principle of data hiding is abandoned when the `friend` keyword is used.

In this chapter the topic of friendship among classes is not discussed. Situations in which it is natural to use friendship among classes are discussed in chapters 17 and 21 and such situations are natural extensions of the way friendship is handled for functions.

There should be a well-defined conceptual reason for declaring friendship (i.e., using the `friend` keyword). The traditionally offered definition of the class concept usually looks something like this:

A class is a set of data together with the functions that operate on that set of data.

As we've seen in chapter 11 some functions have to be defined outside of a class interface. They are defined outside of the class interface to allow promotions for their operands or to extend the facilities of existing classes not directly under our control. According to the above traditional definition of the class concept those functions that cannot be defined in the class interface itself should nevertheless be considered functions belonging to the class. Stated otherwise: if permitted by the language's syntax they would certainly have been defined inside the class interface. There are two ways to implement such functions. One way consists of implementing those functions using available public member functions. This approach was used, e.g., in section 11.2. Another approach applies the definition of the class concept to those functions. By stating that those

functions in fact belong to the class they should be given direct access to the data members of objects. This is accomplished by the `friend` keyword.

As a general principle we state that all functions operating on the data of objects of a class that are declared in the same file as the class interface itself belong to that class and may be granted direct access to the class's data members.

15.1: Friend functions

In section 11.2 the insertion operator of the class `Person` (cf. section 9.3) was implemented like this:

```
ostream &operator<<(ostream &out, Person const &person)
{
   return
      out <<
         "Name:    " << person.name() <<  ", "
         "Address: " << person.address() <<  ", "
         "Phone:   " << person.phone();
}
```

`Person` objects can now be inserted into streams.

However, this implementation required three member functions to be called, which may be considered a source of inefficiency. An improvement would be reached by defining a member `Person::insertInto` and let `operator<<` call that function. These two functions could be defined as follows:

```
std::ostream &operator<<(std::ostream &out, Person const &person)
{
   return person.insertInto(out);
}
std::ostream &Person::insertInto(std::ostream &out)
{
   return
      out << "Name:    " << d_name << ", "
           "Address: " << d_address << ", "
           "Phone:   " << d_phone;
}
```

As `insertInto` is a member function it has direct access to the object's data members so no additional member functions must be called when inserting `person` into `out`.

The next step consists of realizing that `insertInto` is only defined for the benefit of `operator<<`, and that `operator<<`, as it is declared in the header file containing `Person`'s class interface should be considered a function belonging to the class `Person`. The member `insertInto` can therefore be omitted when `operator<<` is declared as a friend.

Friend functions must be declared as friends in the class interface. These *friend declarations* are not *member* functions, and so they are independent of the class's `private, protected` and `public` sections. Friend declaration may be placed anywhere in the class interface. Convention

dictates that friend declaractions are listed directly at the top of the class interface. The class `Person`, using `friend` declaration for its extraction and insertion operators starts like this:

```
class Person
{
   friend std::ostream &operator<<(std::ostream &out, Person &pd);
   friend std::istream &operator>>(std::istream &in, Person &pd);

   // previously shown interface (data and functions)
};
```

The insertion operator may now directly access a `Person` object's data members:

```
std::ostream &operator<<(std::ostream &out, Person const &person)
{
   return
      cout << "Name:    " << person.d_name << ", "
              "Address: " << person.d_address << ", "
              "Phone:   " << person.d_phone;
}
```

Friend declarations are true declarations. Once a class contains friend declarations these friend functions do not have to be declared again below the class's interface. This also clearly indicates the class designer's intent: the friend functions are declared by the class, and can thus be considered functions belonging to the class.

15.2: Extended friend declarations

C++ has added *extended friend declarations* to the language. When a class is declared as a friend, then the `class` keyword no longer has to be provided. E.g.,

```
class Friend;                // declare a class
typedef Friend FriendType;   // and a typedef for it
using FName = Friend;        // and a using declaration

class Class1
{
   friend Friend;            // FriendType and FName: also OK
};
```

In the pre-C++11 standards the friend declaration required an explicit `class`; e.g., `friend class Friend`.

The explicit use of `class` remains required if the compiler hasn't seen the friend's name yet. E.g.,

```
class Class1
{
   // friend Unseen;         // fails to compile: Unseen unknown.
   friend class Unseen;      // OK
};
```

Section 22.10 covers the use of extended friend declarations in class templates.

- Table of Contents

- Previous Chapter
- Next Chapter