Friend function

2.1 Introduction

This chapter continues the discussion of the class begun in Lecture 1. It discusses friend functions, overloading constructors, passing objects to functions, andreturning objects. It also examines a special type of constructor, called the copy constructor, which is used when a copy of an object is needed. The chapter concludes with a description of the this keyword.

2.2 Friend Functions

It is possible to allow a non-member function access to the private members of a class by declaring it a friend of the class. To make a function a friend of a class, include its prototype in the public section of the class declaration and precede it with the friend keyword. For example, in this fragment frnd( ) is declared to be a friend of the class cl:

```
class cl
{
// ... public:
friend void frnd(cl ob);
};
```

The friend keyword gives a non- member function access to the private membersof a class.As you can see, the keyword friend precedes the rest of the prototype. A function may be a friend of more than one class.Here is a short example that uses a friend function to access the private members of myclass:

```
// Demonstrate a friend function.
#include <iostream.h>
class myclass
{
int a, b;
public:
    myclass(int i, int j) { a=i; b=j; }
```

```
        friend int sum(myclass x);
        // sum() is a friend of myclass
    };
 // Note: sum() is not a member function of any class.
 int sum(myclass x)
{
    /* Because sum() is a friend of myclass,
    it can directly access a and b. */
    return x.a + x.b;
}
    int main()
    {
    myclass n(3, 4);
    cout<< sum(n);
    return 0;
    }
```

In this example, the sum( ) function is not a member of myclass. However, it still has full access to the private members of myclass. Specifically, it can access x.a and x.b. Notice also that sum( ) is called normally—not in conjunction with an object and the dot operator. Since it is not a member function, it does not need to be qualified with an object's name. (In fact, it cannot be qualified with an object.) Typically, a friend function is passed one or more objects of the class for which it is a friend, as is the case with sum( ).

While there is nothing gained by making sum( ) a friend rather than a member function of myclass, there are some circumstances in which friend functions are quite valuable. First, friends can be useful for overloading certain types of operators. Second, friend functions simplify the creation of some types of I/O functions. Both of these uses are discussed later in this course.

Thethird reason that friend functions may be desirable is that, in some cases, two or more classes may contain members that are interrelated relative to other parts of your program. For example,

imagine two different classes that each display a pop-up message on the screen when some sort of event occurs. Other parts of your program that are designed to write to the screen will need to know whether the pop-up message is active, so that no message is accidentally overwritten. It is possible to create a member function in each class that returns a value indicating whether a message is active or not; however, checking this condition involves additional overhead (i.e., two function calls, not just one). If the status of the pop-up message needs to be checked frequently, the additional overhead may not be acceptable. However, by using a friend function, it is possible to directly check the status of each object by calling only one function that has access to both classes. In situations like this, a friend function helps you write more efficient code. The following program illustrates this concept.

```
// Use a friend function.
#include <iostream.h>
constint IDLE=0;
constint INUSE=1;
class C2;    // forward declaration
class C1
{
int status;  // IDLE if off, INUSE if on screen
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};
class C2
{int status; // IDLE if off, INUSE if on screen
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};
void C1::set_status(int state)
{
status = state;
}
```

```
void C2::set_status(int state)
{
status = state;
}
// idle( ) is a friend of C1 and C2.
int idle(C1 a, C2 b)
{
   if(a.status || b.status) return 0;
   else return 1;
}

int main()
{
     C1 x; C2 y;
     x.set_status(IDLE);
     y.set_status(IDLE);

     if(idle(x, y)) cout<< "Screen Can Be Used.\n";
     else cout<< "Pop-up In Use.\n";

     x.set_status(INUSE);

     if(idle(x, y)) cout<< "Screen Can Be Used.\n";
     else cout<< "Pop-up In Use.\n";

     return 0;
}
```

The output produced by this program is shown here:

```
Screen Can Be Used.
Pop-up In Use.
```

Because idle( ) is a friend of both C1 and C2 it has access to the private status memberdefined by both classes. Thus, a single call to idle() can simultaneously check the status of an object of each class.

NOTE: A forward declaration declares a class type-name prior to the definition of the class.

Notice that this program uses a forward declaration (also called a forward reference) for the class C2. This is necessary because the declaration of idle( ) inside C1 refers to C2 before it is declared. To create a forward declaration to a class, simply use the form shown in this program.A friend of one class can be a member of another. For example, here is the preceding program rewritten so that idle( ) is a member of C1. Notice the use of the scope resolution operator when declaring idle( ) to be a friend of C2.

```
/* A function can be a member of one class and a
friend of another. */
#include <iostream.h>
constint IDLE=0;
constint INUSE=1;
class C2;    // forward declaration
class C1
{
     int status;  // IDLE if off, INUSE if on screen
public:
void set_status(int state);
     int idle(C2 b);  // now a member of C1
};
class C2
{
     int status;  // IDLE if off, INUSE if on screen
  public:
     void set_status(int state);
     friend int C1::idle(C2 b); // a friend, here
};
void C1::set_status(int state)
{
status = state;
}
void C2::set_status(int state)
{
status = state;
}
```

```
// idle() is member of C1, but friend of C2.
int C1::idle(C2 b)
{
   if(status || b.status) return 0;
   else return 1;
}
int main()
{
   C1 x; C2 y;
   x.set_status(IDLE);
   y.set_status(IDLE);
   if(x.idle(y)) cout<< "Screen Can Be Used.\n";
   else cout<< "Pop-up In Use.\n";

   x.set_status(INUSE);

   if(x.idle(y)) cout<< "Screen Can Be Used.\n";
   else cout<< "Pop-up In Use.\n";

   return 0;
}
```

Since idle( ) is a member of C1, it can access the status variable of objects of typeC1 directly. Thus, only objects of type C2 need be passed to idle( ).