FRIEND FUNCTIONS IN CPP

Friend Functions

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}
// 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;
n.set_ab(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 its private members. Also, notice that **sum( )** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name. Although 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 when you are overloading certain types of operators (see Chapter 14). Second, **friend** functions make the creation of some types of I/O functions easier (see Chapter 18). The third 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. Let's examine this third usage now. To begin, imagine two different classes, each of which displays a pop-up message on the screen when error conditions occur. Other parts of your program may wish to know if a pop-up message is currently being displayed before writing to the screen so that no message is accidentally overwritten. Although you can create member functions in each class that return a value indicating whether a message is active, this means additional overhead when the condition is checked (that is, two function calls, not just one). If the condition needs to be checked frequently, this additional overhead may not be acceptable. However, using a function that is a **friend** of each class, it is possible to check the status of each object by calling only this one function. Thus, in situations like this, a **friend** function allows you to generate more efficient code.

The following program illustrates this concept:

```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int 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;
}
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 << "In use.\n";
x.set_status(INUSE);
if(idle(x, y)) cout << "Screen can be used.\n";
else cout << "In use.\n";
return 0;
}
```

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 may be a member of another.

For example, here is the preceding program rewritten so that **idle( )** is a member of **C1**:

```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int 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);
};
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 << "In use.\n";
x.set_status(INUSE);
if(x.idle(y)) cout << "Screen can be used.\n";
else cout << "In use.\n";
return 0;
}
```

Because **idle( )** is a member of **C1**, it can access the **status** variable of objects of type **C1** directly. Thus, only objects of type **C2** need be passed to **idle( )**. There are two important

restrictions that apply to **friend** functions. First, a derived class does not inherit **friend** functions. Second, **friend** functions may not have a storage-class specifier. That is, they may not be declared as **static** or **extern.**

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf