APPENDIX 7

The this Pointer

When defining member functions for a class, you sometimes want to refer to the calling object. The *this* pointer is a predefined pointer that points to the calling object. For example, consider a class like the following:

```
class Sample
{
public:
        ...
    void show_stuff();
        ...
private:
    int stuff;
        ...
};
```

The following two ways of defining the member function show_stuff are equivalent:

```
void Sample::show_stuff()
{
    cout << stuff;
}
//Not good style, but this illustrates the this pointer:
void Sample::show_stuff()
{
    cout << (this->stuff);
}
```

Notice that *this* is not the name of the calling object, but is the name of a pointer that points to the calling object. The *this* pointer cannot have its value changed; it always points to the calling object.

As the comment before the previous sample use of *this* indicates, you normally have no need for the pointer *this*. However, in a few situations it is handy.

Overloading the assignment operator

One place where the *this* pointer is commonly used is in overloading the assignment operator =. For example, consider the following class:

```
class StringClass
{
public:
    ...
    StringClass& operator =(const StringClass& right_side);
    ...
private:
    char *a;//Dynamic array for a string value ended with '\0.'
};
```

The following definition of the overloaded assignment operator can be used in chains of assignments like

```
s1 = s2 = s3;
```

This chain of assignments means

```
s1 = (s2 = s3);
```

The definition of the overloaded assignment operator uses the *this* pointer to return the object on the left side of the = sign (which is the calling object):

```
//This version does not work in all cases. Also see the next version.
StringClass& StringClass::operator =(const StringClass& right_side)
{
    delete [] a;
    a = new char[strlen(right_side.a) + 1];
    strcpy(a, right_side.a);
    return *this;
}
```

The definition above does have a problem in one case: If the same object occurs on both sides of the assignment operator (like `s=s;`), then the array member will be deleted. To avoid this problem, you can use the *this* pointer to test this special case as follows:

```
//Final version with bug fixed:
StringClass& StringClass::operator =(
const StringClass& right_side)
{
    if (this == &right_side)
    {
        return *this;
    }
    else
    {
        delete [] a;
        a = new char [strlen(right_side.a) + 1];
        strcpy(a, right_side.a);
        return *this;
    }
}
```

In the section of Chapter 11 entitled "Overloading the Assignment Operator," we overloaded the assignment operator for a string class called `StringVar`. In that section, we did not need the *`this`* pointer because we had a member variable called `max_length` that we could use to test whether or not the same object was used on both sides of the assignment operator `=`. With the class `StringClass` discussed above, we have no such alternative because there is only one member variable. In this case, we have essentially no alternative but to use the *`this`* pointer.