SESSION

28

The Assignment Operator

Session Checklist

✔ Introduction to the assignment operator

✔ Why and when the assignment operator is necessary

✔ Similarities between the assignment operator and the copy constructor

30 Min.
To Go

Whether or not you start out overloading operators, you need to learn to overload the assignment operator fairly early. The assignment operator can be overloaded for any user-defined class. By following the pattern provided in this session, you will be generating your own version of `operator=()` in no time.

Why Is Overloading the Assignment Operator Critical?

C++ provides a default definition for `operator=()` for all user-defined classes. This default definition performs a member-by-member copy, like the default copy constructor. In the following example, each member of `source` is copied over the corresponding member in `destination`.

```
void fn()
{
   MyStruct source, destination;
   destination = source;
}
```

However, this default definition is not correct for classes that allocate resources, such as heap memory. The programmer must overload `operator=()` to handle the transfer of resources.

Comparison with copy constructor

The assignment operator is much like the copy constructor. In use, the two look almost identical:

```
void fn(MyClass &mc)
{
    MyClass newMC(mc);   // of course, this uses the
                         // copy constructor
    MyClass newerMC = mc;// less obvious, this also invokes
                         // the copy constructor
    MyClass newestMC;    // this creates a default object
    newestMC = mc;       // and then overwrites it with
                         // the argument passed
}
```

The creation of `newMC` follows the standard pattern of creating a new object as a mirror image of the original using the copy constructor `MyClass(MyClass&)`. Not so obvious is that C++ allows the second format in which `newerMC` is created using the copy constructor.

However, `newestMC` is created using the default (void) constructor and then overwritten by `mc` using the assignment operator. The difference is that when the copy constructor was invoked on `newerMC`, the object `newerMC` did not already exist. When the assignment operator was invoked on `newestMC`, it was already a `MyClass` object in good standing.

Tip

The rule is this: The copy constructor is used when a new object is being created. The assignment operator is used if the left-hand object already exists.

Like the copy constructor, an assignment operator should be provided whenever a shallow copy is not appropriate. (Session 20 has a full discussion of shallow versus deep constructors.) It suffices to say that a copy constructor and an assignment operator should be used when the class allocates and saves resources within the class so that you don't end up with two objects pointing to the same resource.

20 Min. To Go

How Do I Overload the Assignment Operator?

Overloading the assignment operator is similar to overloading any other operator. For example, Listing 28-1 is the program `DemoAssign`, which includes both a copy constructor and an assignment operator.

Note

Remember that the assignment operator must be a member function of the class.

Listing 28-1
Overloading the Assignment Operator

```
// DemoAssign - demonstrate the assignment operator
#include <stdio.h>
#include <string.h>
#include <iostream.h>

// Name - a generic class used to demonstrate
//        the assignment and copy constructor
//        operators
class Name
{
  public:
    Name(char *pszN = 0)
    {
        copyName(pszN);
    }
    Name(Name& s)
    {
        copyName(s.pszName);
```

Continued

Listing 28-1 *Continued*

```
    }
    ~Name()
    {
        deleteName();
    }
    //assignment operator
    Name& operator=(Name& s)
    {
        //delete existing stuff...
        deleteName();
        //...before replacing with new stuff
        copyName(s.pszName);
        //return reference to existing object
        return *this;
    }

    // display - output the current object
    //           the default output object
    void display()
    {
        cout << pszName;
    }

  protected:
    void copyName(char *pszN);
    void deleteName();
    char *pszName;
};

// copyName() - allocate heap memory to store name
void Name::copyName(char *pszName)
{
    this->pszName = 0;
    if (pszName)
    {
        this->pszName = new char[strlen(pszName) + 1];
        strcpy(this->pszName, pszName);
    }
```

```
}

// deleteName() - return heap memory
void Name::deleteName()
{
    if (pszName)
    {
        delete pszName;
        pszName = 0;
    }
}

// displayNames - output function to reduce the
//                number of lines in main()
void displayNames(Name& pszN1, char* pszMiddle,
                  Name& pszN2, char* pszEnd)
{
    pszN1.display();
    cout << pszMiddle;
    pszN2.display();
    cout << pszEnd;
}

int main(int nArg, char* pszArgs[])
{
    // create two objects
    Name n1(“Claudette”);
    Name n2(“Greg”);
    displayNames(n1, “ and “,
                 n2, “ are newly created objects\n”);

    // now make a copy of an object
    Name n3(n1);
    displayNames(n3, “ is a copy of “,
                 n1, “\n”);

    // make a copy of the object from the
    // address
```

Continued

Listing 28-1 *Continued*

```
        Name* pN = &n2;
        Name n4(*pN);
        displayNames(n4, " is a copy using the address of ",
                     n2, "\n");

        // overwrite n2 with n1
        n2 = n1;
        displayNames(n1, " was assigned to ",
                     n2, "\n");
        return 0;
    }
```

Output:

```
Claudette and Greg are newly created objects
Claudette is a copy of Claudette
Greg is a copy using the address of Greg
Claudette was assigned to Claudette
```

The class `Name` retains a person's name in memory, which it allocates from the heap in the constructor. The constructors and destructor for class `Name` are similar to those presented in Sessions 19 and 20. The constructor `Name(char*)` copies the name given it to the `pszName` data member. This constructor also serves as the default constructor. The copy constructor `Name(&Name)` copies the name of the object passed to the name stored in the current object by calling `copyName()`. The destructor returns the `pszName` character string to the heap by calling `deleteName()`.

The function `main()` demonstrates each of these member functions. The output from `DemoAssign` is shown at the end of Listing 28-1 above.

Take an extra look at the assignment operator. The function `operator=()` looks to all the world like a destructor immediately followed by a copy constructor. This is typical. Consider the assignment in the example `n2 = n1`. The object `n2` already has a name associated with it ("Greg"). In the assignment, the memory that the original name occupies must be returned to the heap by calling `deleteName()` before new memory can be allocated into which to store the new name ("Claudette") by calling `copyName()`.

The copy constructor did not need to call `deleteName()` because the object didn't already exist. Therefore, memory had not already been assigned to the object when the constructor was invoked.

In general, an assignment operator has two parts. The first part resembles a destructor in that it deletes the assets that the object already owns. The second part resembles a copy constructor in that it allocates new assets.

Two more details about the assignment operator

There are two more details about the assignment operator of which you need to be aware. First, the return type of `operator=()` is `Name&`. I didn't go into detail at the time, but the assignment operator is an operator like all others. Expressions involving the assignment operator have both a value and a type, both of which are taken from the final value of the left-hand argument. In the following example, the value of `operator=()` is 2.0 and the type is `double`.

```
double d1, d2;
void fn(double );
d1 = 2.0;
```

This is what enables the programmer to write the following:

```
d2 = d1 = 2.0
fn(d2 = 3.0);    // performs the assignment and passes the
                 // resulting value to fn()
```

The value of the assignment `d1 = 2.0`, 2.0, and type, `double`, are passed to the next assignment operator. In the second example, the value of the assignment `d2 = 3.0` is passed to the function `fn()`.

I could have made `void` the return type of `Name::operator=()`. However, if I did, the above example would no longer work:

```
void otherFn(Name&);
void fn()
{
   Name n1, n2, n3;

   // the following is only possible if the assignment
   // operator returns a reference to the current object
   n1 = n2 = n3;
   otherFn(n1 = n2);
}
```

The results of the assignment `n1 = n2` is `void`, the return type of `operator=()`, which does not match the prototype of `otherFn()`. Declaring `operator=()` to return a reference to the “current” object and returning `*this` retains the semantics of the assignment operator for intrinsic types.

The second detail is that `operator=()` was written as a member function. Unlike other operators, the assignment operator cannot be overloaded with a nonmember function. The special assignment operators, such as += and *=, have no special restrictions and can be nonmember functions.

10 Min. To Go

An Escape Hatch

Providing your class with an assignment operator can add considerable flexibility to the application code. However, if this is too much for you, or if you can’t make copies of your object, overloading the assignment operator with a protected function will keep anyone from accidentally making an unauthorized shallow copy. For example:

```
class Name
{
  //...just like before...
  protected:
    //assignment operator
    Name& operator=(Name& s)
    {
       return *this;
    }
};
```

With this definition, assignments such as the following are precluded:

```
void fn(Name &n)
{
    Name newN;
    newN = n;       //generates a compiler error -
                    //function has no access to op=()
}
```

Done!

This copy protection for classes saves you the trouble of overloading the assignment operator but reduces the flexibility of your class.

Tip

If your class allocates resources such as memory off of the heap you *must* either write a satisfactory assignment operator and copy constructor or make both protected to preclude the default provided by C++ from being used.

REVIEW

Assignment is the only operator that you must overload and then only under certain conditions. Fortunately, defining assignment for your class isn't too difficult if you follow the pattern laid out for you in this session.

- C++ provides a default assignment operator that performs member-by-member copies. This version of assignment is fine for many class types; however, classes that can be allocated resources must include a copy constructor and an overloaded assignment operator.
- The semantics of the assignment operator is generally similar to a destructor immediately followed by a copy constructor. The destructor removes whatever resources might already be in the class, while the copy constructor makes a deep copy of the resources assigned it.
- Declaring the assignment operator protected removes the danger but limits the class by precluding assignment to your class.

QUIZ YOURSELF

1. When do you need to include an assignment operator in your class? (See "Why is Overloading the Assignment Operator Critical?")
2. The return type of the assignment operator should always match the class type. Why? (See "Two More Details About the Assignment Operator.")
3. How can you avoid the need to write an assignment operator? (See "An Escape Hatch.")