Copy Constructor and operator=

We already know that the compiler will supply a default (zero-argument) constructor if the programmer does not specify one. This default constructor will call each data member's default constructor in order to initialize the object. If the programmer wishes to override that default constructor, he or she simply provides one.

There is another type of constructor the compiler generates — it is called a **copy constructor**, and it's called whenever an object needs to be constructed from an existing one. Suppose we had a class to encapsulate strings, and we call this class **`mystring`**[1] . It might include such data members as the length of the string as well as the characters that would be in the string.

```
class mystring
{
   public:
      mystring(const char* s = "");
      ~mystring();

   ...
   private:
      int length;
      char *str;
};
```

For this example, we'll assume that the **`mystring`** constructor will allocate space for the characters, and the destructor will free that space.

The copy constructor may be called when doing simple initializations of a **`mystring`** object:

```
mystring me("Geraldo");
mystring clone = me;     // copy constructor gets called.
```

More importantly, the copy constructor is called when passing an object by value, or returning an object by value. For instance, we might have a function which opens a file:

```
static void openFile(mystring filename)
{
   // Convert the object to a character string, open a stream...
}
```

[1] We're calling it **`mystring`** so we don't get confuse it with the built-in string class.

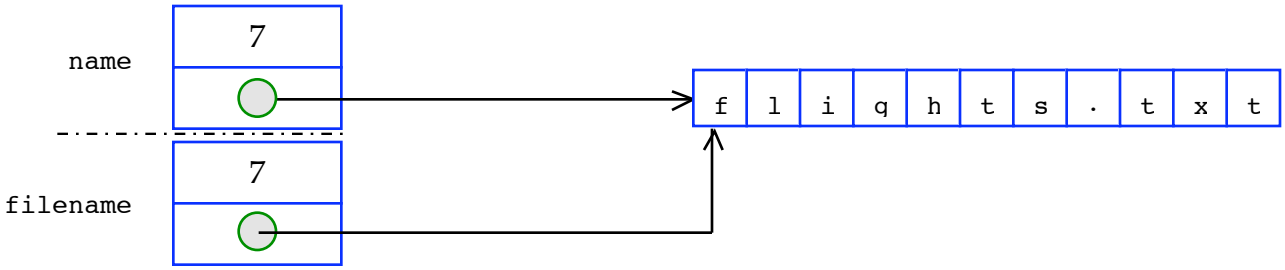
We might declare a string and call the **`OpenFile`** function like this:

```
mystring name("flights.txt");
openFile(name);
```

When passing the name object, the copy constructor is called to copy the **`mystring`** object from the calling function to the local parameter in the **`openFile`** function. Because we did not specify a copy constructor, the default copy constructor is called.

Default Copy Constructor

The default copy constructor does a member-wise copy of an object. For our **`mystring`** class, the default copy constructor will copy the length integer and the characters pointer. However, the characters themselves are not copied. This is called a **shallow copy**, because it only copies the data one level deep in a class hierarchy. In memory, what we'd have would look like this:

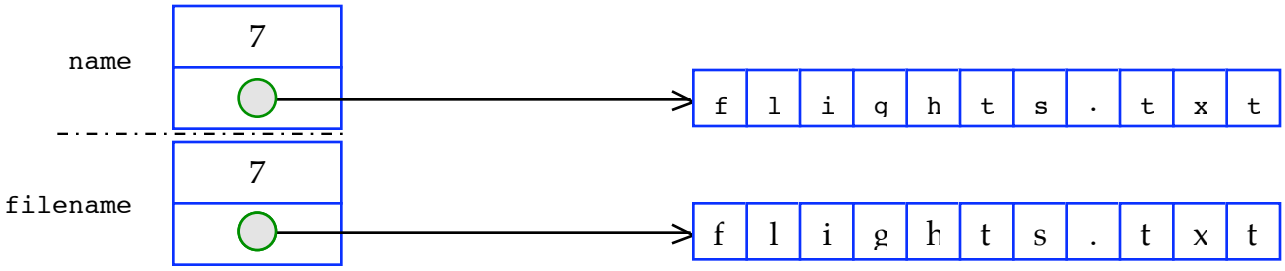


This causes a problem because now the two objects share the same characters data. If we changed any of the characters within the `filename` object, it would change the characters in the **`name`** as well. What's worse is that after the **`OpenFile`** function exits, the **`mystring`** destructor is called on the **`filename`** object, which frees the characters data member. Now the **`name`** object has an invalid characters data member!

What We Want: The Deep Copy

In order to avoid potential disasters like this, what we want is a **deep copy** of the **`mystring`** object. We want a copy of the length, and we want a copy of the entire **`str`** data member — not just a copy of the pointer.

What we want is a picture in memory which looks like this:



Now we can manipulate the characters within the filename object and not affect the original name object. When the character array within the copy is freed on exit the **`openFile`** function, the original **`mystring`** object is still going strong.

Declaring a Copy Constructor

In order to provide deep-copy semantics for our **`mystring`** object, we need to declare our own copy constructor. A copy constructor takes a constant reference to an object of the class' type as a parameter and has no return type. We would declare the copy constructor for the **`mystring`** within the class declaration like this:

```
class mystring
{
   public:
      mystring(const char* s = "");
      mystring(const mystring& s);
      ...
};
```

We might implement the copy constructor like this:

```
mystring::mystring(const mystring& s)
{
   length = s.length;
   str = new char[length + 1];
   strcpy(str, s.str);
}
```

You should provide a copy constructor for any class for which you want deep copy semantics. If the class only has data members that are integral types (that is, it contains no pointers or open files) and/or direct objects (which have properly implemented copy constructors), you can omit the copy constructor and let the default copy constructor handle it.

Limitations

The copy constructor is not called when doing an object-to-object assignment. For instance, if we had the following code, we would still only get a shallow copy:

```
mystring betty("Betty Rubble");  // Initializes string to "Betty Rubble"
mystring bettyClone;             // Initializes string to empty string
bettyClone = betty;
```

This is because the **assignment operator** is being called instead of the copy constructor. By default, the assignment operator does a member-wise copy of the object, which in this case gives a shallow copy. However, C++ gives us the ability to override the default assignment operator, and we'll learn that today too.

Incidentally, there may be cases when you want to prevent anyone from copying a class object via the copy constructor. By declaring a copy constructor as a **`private`** constructor within the class definition and **not** implementing it, the compiler will prevent the passing of objects of that class by value.

Assignment

It is possible to redefine the **assignment** operator for a class. The assignment operator must be defined as a member function in order to guarantee that the left-hand operand is an object. By default, the assignment operator is defined for every class, and it does a member-wise copy from one object to another. This is a shallow copy, so the assignment operator should generally be redefined for any class that contains pointers. You can declare the assignment operator as a **`private`** operator and not provide an implementation in order to prevent object-to-object assignment.

The key difference between assignment and copy construction is that assignment possibly involves the destruction of embedded resources. We need to overload the assignment operator whenever we want to guarantee that memory (or perhaps other resources) aren't orphaned improperly. The syntax for the assignment method is a trifle quirky, but it's just that: syntax, and you just treat the prototype as boilerplate notation and otherwise implement the method as you would any other.

```
const mystring& operator=(const mystring& rhs)
{
   if (this != &rhs) {
      delete[] this->str;  // donate back useless memory
      this->str = new char[strlen(rhs.str) + 1];  // allocate new memory
      strcpy(this->str, rhs.str); // copy characters
      this->length = rhs.length; // copy length
   }

   return *this;  // return self-reference so cascaded assignment works
}
```

A good way to look at the assignment operator: Think of it as destruction followed by immediate reconstruction. The **`delete[]`** would appear in the destructor as well, whereas the next three lines are very constructor-ish.

Caveats 1: The **`this != &rhs`** checks to see whether we're dealing with a self-assignment. An expression of the form **`name = name`** is totally legal, but there's no benefit in actually destroying the object only to reconstruct it only to take the same form. In fact, the **`delete[]`** line would actually release the very memory that we need on the very next line.

Caveat 2: The return type is **`const mystring&`**, and the `return` statement is `return *this`, because cascaded assignments involving **`mystring`**s should behave and propagate right to left just as they would with **`int`**s or **`double`**s.

```
mystring heroine("ginger");
mystring cluck("babs");
mystring quack("rhodes");
mystring meanie("mrs. tweety");

meanie = quack = cluck = heroine;
cout << heroine << cluck << quack << meanie << endl;
// should print: gingergingergingerginger
```