

Assignment-to-Self Overkill

Conrad Weisert, Information Disciplines, Inc., Chicago

May, 1997

An odd bit of programming folklore keeps turning up in C++ textbooks and courses: an admonition to treat *assignment-to-self* as a special case in overloading the assignment operator for non-contiguous objects. The issue typically arises in a dynamic-length character string class with this internal representation:

```
class String{
    int    length;
    char*  value;
    .
    .
};
```

They first show us a naïve version of so-called “deep copy” assignment:

```
String& operator= (const String& rs)
{delete [] value;
 length = rs.length;
 value = new char [length+1];
 strcpy (value, rs.value);
 return *this;
}
```

They then point out correctly that the above version will fail catastrophically whenever the left and right sides are the same `String` object, as they might be once during a loop like this (when `ctr = k`):

```
for (ctr = 0; ctr < saSize; ctr++)
    sa[ctr] = sa[k];
```

They offer this correction:

```
String& operator= (const String& rs)
{if (this == &rs) return *this;
 .
 .
    (the rest as in the original)
 .
}
```

Myers¹ advises “Check for assignment to self in operator=”, while Coplien² offers the same solution as part of his “Orthodox Canonical Class Form”. Johnsonbaugh & Kalin³, Lippman⁴, and Perry⁵ present the same solution, while Eckel generalizes the principle:

“ . . . when defining `operator=` you must perform the duties of the destructor on the current object before performing the duties of the copy constructor with the argument to the right of the equal sign.”⁶

(These are all useful books that I recommend enthusiastically. If this example is a lapse, it’s a minor one that doesn’t detract from the authors’ contributions.)

Not so! The straightforward solution is just to check whether the string’s *length* is being changed:

```
String& operator= (const String& rs)
{if (length != rs.length)
    {delete [] value;
     length = rs.length;
     value = new char [length+1];
    }
 strcpy (value, rs.value);
 return *this;
}
```

The *main* flaw in the original naïve solution wasn’t the assignment-to-self failure. It was the unnecessary freeing and reallocating of memory when the string’s length isn’t being changed. In correcting that shortcoming we eliminate the need to handle assignment to self as a special case⁷.

Our simple solution is also much more *efficient* whenever the source and the target have the same length, a common occurrence in business applications (*far* more common than assigning a string to itself). Freeing and allocating memory are expensive operations that we should avoid⁸ whenever we can.

Are there *any* situations where the assignment operator should either (a) “check for assignment to self” or (b) free and reallocate resources? Indeed there are, but this amazingly popular example isn’t one of them.

¹ Scott Myers: *Effective C++*, Addison-Wesley, ISBN 0-201-56364-9, item 17.

² James Coplien: *Advanced C++ Programming Styles & Idioms*, Addison Wesley, ISBN 0-201-54855-0, p. 41.

³ Richard Johnsonbaugh and Martin Kalin: *Object-Oriented Programming in C++*, Prentice Hall, ISBN 0-02-360682-7, p. 154.

⁴ Stanley B. Lippman, *C++ Primer*, Addison Wesley, ISBN 0-201-54848-8, p. 319.

⁵ Greg Perry et al: *Killer Borland C++*, Que, ISBN 1-56529-685-0, p. 486.

⁶ Bruce Eckel: *C++ Inside and Out*, Osborne, ISBN 0-07-881809-5, p. 475.

⁷ We may still choose to do so if we’re worried about the *time* required to copy a long string to itself, but that’s a different issue that affects only the `strcpy` call, not memory management.

⁸ The *Standard Template Library* uses more sophisticated techniques to minimize memory management, even when the size is being changed.