Copy Assignment Operator Overload

The material in this handout is collected from the following references:

- Section 13.1 of the text book C++ Primer.
- Various sections of Effective C++.
- Various sections of More Effective C++.
- Microsoft has additional information and examples on copy constructors and copy assignments.

Return type of assignment operator

The C++ standard went to a lot of trouble to ensure that user-defined types would mimic built-in types as closely as possible. That's why you can overload operators, write type conversion functions, and take control of copy construction and assignment. With built-in types, you can chain assignments together, like so:

```
int w, x, y, z;
w = x = y = z = 0;
```

As a result, you should be able to chain together assignments for user-defined types, too:

```
std::string w, x, y, z;
w = x = y = z = "Hello";
```

The assignment operator is right-associative, so the assignment chain is parsed like this:

```
w = (x = (y = (z = "Hello")));
```

Writing this in its equivalent functional form

```
w.operator=(x.operator=(y.operator=(z.operator=("Hello"))));
```

This functional form emphasizes that the argument to `w.operator=`, `x.operator=`, and `y.operator=` is the return value of a previous call to `operator=`. As a result, the return type of `operator=` must be acceptable as an input to the function itself. For the default version of `operator=` in a class `C`, the signature of the function is as follows

```
C& C::operator=(const C&);
```

You'll almost always want to follow this convention of having `operator=` both take and return a reference to a class object, although at times you may overload `operator=` so that it takes different argument types. For example, the `std::string` type provides two different versions of the copy assignment:

```
std::string& operator=(std::string const& rhs); // assign a string to a string
std::string& operator=(char const *rhs);        // assign a char* to a string
```

Notice, however, that even in the presence of overloading, the return type is a reference to an object of the class.

A common error amongst new C++ programmers is to have `operator=` return `void`, a decision that seems reasonable until you realize it prevents chains of assignment. So don't do it.

Another common error is to have `operator=` return a reference to a `const` object, like this:

```
class Widget {
public:
  const Widget& operator=(const Widget& rhs); // note const return
  // other stuff not of interest
};
```

The usual motivation is to prevent clients from doing weird things like this:

```
Widget w1, w2, w3;
// assign w2 to w1, then w3 to the result!!! Giving Widget's
// operator= a const return value prevents this from compiling.
(w1 = w2) = w3;
```

Although this looks weird, it is not prohibited for the built-in types:

```
int i1, i2, i3;
(i1 = i2) = i3; // legal! assigns i2 to i1, then i3 to i1!
```

Although the above expression has no practical use, if it's good enough for the `int`s, it should be good enough for you and your classes.

Which object to return?

Within a copy assignment bearing the default signature, there are two obvious candidates for the object to be returned: the object on the left hand side of the assignment [the one pointed to by `this`] and the object on the right-hand side [the one named in the parameter list]. Which is correct?

Here are the possibilities for a `String` class:

```
String& String::operator=(const String& rhs) {
  // function definition
  return *this; // return reference to left-hand object
}

String& String::operator=(const String& rhs) {
  // function definition
  return rhs; // return reference to right-hand object
}
```

There are important differences between the two.

First, the version returning `rhs` won't compile. That's because `rhs` is a reference-to-`const`-`String`, but `operator=` returns a reference-to-`String`. That seems easy enough to get around - just redeclare `operator=` like this:

```
String& String::operator=(String& rhs)   { ... }
```

But now client code won't compile! Look again at the last part of the original chain of assignments:

```
x = "Hello"; // same as x.op=("Hello");
```

Because the right-hand argument of the assignment is not of the correct type - it's a `char` array, not a `String` — compilers would have to create a temporary `String` object via the `String` constructor to make the call succeed. That is, they'd have to generate code roughly equivalent to this:

```
const String temp{"Hello"}; // create temporary
x = temp;                   // pass temporary to op=
```

Compilers are willing to create such a temporary [unless the needed constructor is `explicit`], but note that the temporary object is `const`. This is important, because it prevents you from accidentally passing a temporary into a function that modifies its parameter. If that were allowed, programmers would be surprised to find that only the compiler-generated temporary was modified, not the argument they actually provided at the call site.

Now we can see why the client code above won't compile if `String`'s `operator=` is declared to take a reference-to-non-`const` `String`: it's never legal to pass a `const` object to a function that fails to declare the corresponding parameter `const`. That's just simple `const`-correctness.

You thus find yourself in the happy circumstance of having no choice whatsoever: you'll always want to define your copy assignments in such a way that they return a reference to their left-hand argument, `*this`. If you do anything else, you prevent chains of assignments, you prevent implicit type conversions at call sites, or both.

Check for assignment to self

An assignment to self occurs when you do something like this:

```
class X { ... };
X a;
a = a; // a is assigned to itself
```

It is perfectly legal to write such self-assigning expressions for built-in types:

```
int i {5};
i = i;
std::cout << "i: " << i << "\n"; // writes 5 to standard output
```

Assignment to self can also appear in other ways:

```
int a[100], i{5}, j{i};
a[i] = a[j]; // assignment to self
int *pi{a+i}, *pj{a+j};
*pi = *pj;   // assignment to self
```

Assignment to self can appear in this more benign-looking form:

```
X& b {a};
a = b;   // a is assigned to itself since b is another name for a
```

If `b` is another name for `a` [for example, a reference that has been initialized to `a`], then this is also an assignment to self, though it doesn't outwardly look like it. This is an example of *aliasing*: having two or more names for the same underlying object. Aliasing is important because it can crop up in any number of nefarious disguises, so you need to take it into account any time you write a function.

Two good reasons exist for taking special care to cope with possible aliasing in copy assignments. The lesser of them is *efficiency*. If you can detect an assignment to self at the top of your copy assignments, you can return right away, possibly saving a lot of work that you'd otherwise have to go through to implement assignment.

A more important reason for checking for assignment to self is to ensure *correctness*. Remember that a copy assignment must typically free the resources allocated to an object [i.e., get rid of its old value] before it can allocate the new resources corresponding to its new value. When assigning to self, this freeing of resources can be disastrous, because the old resources will be needed during the process of allocating the new ones.

Consider assignment of `String` objects, where the copy assignment fails to check for assignment to self:

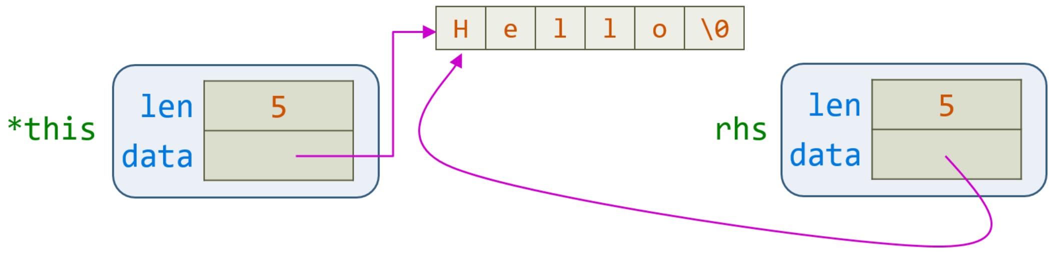
```
class String {
public:
  String(const char *value);
  ~String();
  String& operator=(String const& rhs);
private:
  size_t len;
  char *data;
};

// operator= that omits a check for assignment to self
String& String::operator=(String const& rhs) {
  len = rhs.len; // copy rhs's len value into it
  delete [] data;     // delete old memory
  data = new char[len + 1];     // allocate new memory and
  std::strcpy(data, rhs.data); // copy rhs's value into it
  return *this;
}
```

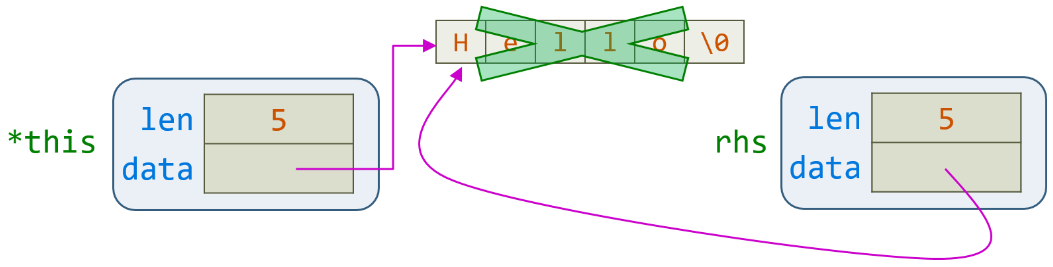
Consider now what happens in this case:

```cpp
String a{"Hello"};
a = a; // same as a.operator=(a)
```

Inside the copy assignment, `*this` and `rhs` seem to be different objects, but in this case they happen to be different names for the same object. You can envision it like this:



The first thing the copy assignment does is use `delete` on `data`, and the result is the following state of affairs:



Now when the copy assignment tries to do a `strcpy` on `rhs.data`, the results are undefined. This is because `rhs.data` was deleted when `data` was deleted, which happened because `data`, `this->data`, and `rhs.data` are all the same pointer! From this point on, things can only get worse.

By now you know that the solution to the dilemma is to check for an assignment to self and to return immediately if such an assignment is detected:

```cpp
// operator= that omits a check for assignment to self
String& String::operator=(String const& rhs) {
  if (this == &rhs) return *this; // if self-assignment, do nothing

  len = rhs.len; // copy rhs's len value into it
  delete [] data;     // delete old memory
  data = new char[len + 1];     // allocate new memory and
  std::strcpy(data, rhs.data); // copy rhs's value into it
  return *this;
}
```

Exception-safe implementation

Although the above copy assignment works, it is exception-unsafe. Notice that the function first deletes the old string and it is possible that an exception is thrown by the memory allocation request for the new string. Such a situation will cause the `String` to end up deleting its previous data without being able to dynamically allocate memory for new data. Making `operator=` exception-safe makes it safe from self-assignments too:

```cpp
// operator= that omits a check for assignment to self
// and also makes it exception-safe
String& String::operator=(String const& rhs) {
  // first make "new" String
  size_t tmp_len {rhs.len};
  char *tmp_data {new char [tmp_len + 1]};
  std::strcpy(tmp_data, rhs.data);

  delete [] rhs.data; // next, delete "old" String

  len  = tmp_len;     // make "new" String permanent
  data = tmp_data;

  return *this;       // return reference to self
}
```

If you're concerned about efficiency, you could put the identity test back at the top of the function. Before doing that, however, ask yourself how often you expect self-assignments to occur, because the test isn't free. It makes the code [both source and object] a bit bigger, and it introduces a branch into the flow of control, both of which can decrease runtime speed. The effectiveness of instruction prefetching, caching, and pipelining can be reduced, for example.

An alternative to manually ordering statements to make sure the implementation is both exception- and self-assignment-safe is to use the *copy and swap* idiom. This idiom consists of the following:

1. Write a destructor that deletes any owned resource.
2. Write a copy constructor that duplicates any owned resource and takes ownership of it.
3. Write a `swap` function that will exchange the contents of two objects by swapping their internal bits.
4. Write the copy assignment by making a temporary copy of the source object, then swap the copy with `*this`.

The copy and swap idiom would look like this:

```
// operator= with copy and swap idiom
String& String::operator=(String const& rhs) {
  String tmp{rhs}; // make a copy of rhs's data
  swap(tmp);       // swap *this's data with rhs
  return *this;    // return reference to self
}
```

An implementation of `String::swap` would look like this:

```
void String::swap(String& rhs) {
  std::swap(len, rhs.len); // std::swap is presented in <algorithm>
  std::swap(data, rhs.data);
}
```