MTH 4300, Lecture 18

`operator=`;
Copy Constructors and the Rule of Three;
Actual Vectors

E Fink

April 7, 2025

1. operator=

Consider the following code, which can be found in **copies.cpp** in **L18x1_List** :

```
List x(10), y(10);
x.append("Hello");
x.append("Goodbye");
y = x; // Let's think about this line
x[0] = "Replace";
```

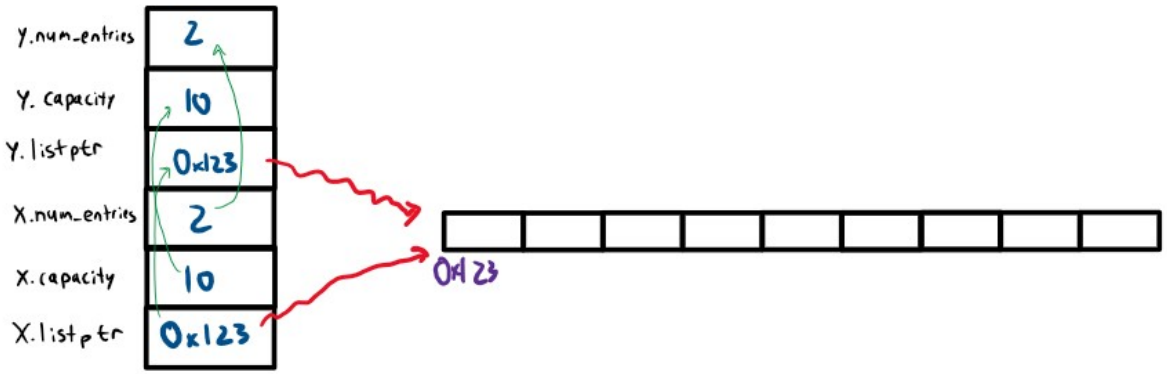
What happens in the line `y = x`?

In fact, `y.operator=(x)` is quietly called. If you haven't implemented `operator=()`, C++ will provide it for you . . . but when your class has pointer members, the default implementation may not do what you want or expect.

operator=

The default implementation of `operator=()` simply copies the values for all member variables. In the case of the last slide, when `y = x` takes place, the value of `x.listptr` is assigned to `y.listptr`.

But that means that the arrays holding the elements of x and y are the same. (This is called *shallow copying*.) So, when you update `x`'s array, you are also updating `y`'s array.



If you want to be able to write assignments like `y = x` where `y` and `x` are independent `Lists` that just happened to have the same value at the time of the assignment (*deep copying*), you can write your own implementation of `operator=` which overrides the default one.

operator=

We implement

`List& operator=(const List&)`

where the return type is a `List&` so that we may perform chained equalities like `a = b = c` without having to create copies of objects.

In the current case, the operator should

- copy over the non-pointer members
- deallocate `listptr`'s current array
- create an entirely new dynamically-allocated array of strings, and assign it to `listptr`
- copy the entries of the right-hand-side object's array one-by-one to the new array
- and return `*this` (to support the chaining).

operator=

When implementing `operator=`, we take special care with the case of self-assignment. There's a surprising danger that we run into if, say, `x` were a `List`, and we wrote

```
x = x;
```

The second bullet from the last slide would `delete[] x.listptr`.

The third bullet would create a new "blank" array for `x.listptr` to point to.

And the fourth bullet would then start filling the new blank array with entries from `...x.listptr`, the very same blank array we just created. So the new array will be blank – it certainly won't be unchanged from its original state, which is what we'd expect.

To avoid this, we put in a special check to ensure that `this` (the address of the left-hand-side object) is different from the address of the right-hand-side. This is a common pattern for assignment operators.

(It's unlikely you'd write `x = x;` exactly, but sneaky self-assignments can find their way into your code.)

2. Copy Constructors and the Rule of Three

A *copy constructor* is a constructor whose declaration looks like

```
MyClass (const MyClass&);
```

(the `const` could be omitted, but usually isn't). This constructor would be called, of course, when one writes a declaration like one of

```
MyClass x(y);
```

where `y` is also a `MyClass` object. However, the copy constructor is also called in other instances, like

- `MyClass z = y; // where y is a MyClass object`
- when a function has a `MyClass` non-reference parameter
- when a function has a `MyClass` non-reference return value.

L18x2_which.cpp

Now, let's add a copy constructor to our `List` class. It should be a simpler version of the copy assignment. (Note: copy constructors will never have the self-assignment problem, because their job is creating brand new objects.)

Copy Constructors and the Rule of Three

The *Rule of Three* says that if you think you have a good reason to implement ONE out of the three of

destructor, assignment operator, copy constructor

then **you almost surely should implement all three**.

It's perfectly ok to implement none of them: C++ will automatically provide all three. But if your class' objects reserve heap memory, then that memory needs to be carefully released by a destructor which you design yourself. When that happens, it's usually the case that when a copy or assignment of an object is made, a copy of the dynamic memory owned by that object needs to be made.

This rule is especially important because it is very easy to not notice when destructors and copy constructors are being called!

L18x3_riddle.cpp

Copy Constructors and the Rule of Three

To unravel the riddle: in `main()`, first, `x` is created on the stack, and `x.arrptr` is assigned the address of a dynamic array.

Then, `x.fn()` is called. This function returns `*this` by value. That means that a COPY of `x` is created in the temporary memory. **Since there is no copy constructor**, this new `GarbageArray` has its value of `arrptr` the same as `x.arrptr`.

On the line of code `x.fn();`, nothing happens to the return value, and when the line has been fully executed, the temporary copy will no longer be needed – so the destructor will be called. But **this destructor will delete** `x.arrptr`!

Now, go down to the second line which calls `x.fn()`. The process repeats itself, except that this time, when you delete `x.arrptr`, you're deleting a list that has already been deallocated – giving an error.

3. Actual Vectors

Finally, let's talk about an actual type that C++ supports, which does what `Lists` try to do: `vectors`! These are part of the *Standard Template Library*, which contains many types of data containers.

To use `vectors`, you `#include <vector>`. To declare a `vector`, write

```
vector<my_type> x; // my_type is any data type
```

or

```
vector<double> x(20, 3.14); // vector with pi 20 times
```

or

```
vector<int> x = {1, 2, 4, 8, 10}; // Initializer list
```

Among other things, vectors support the following methods:

- `.push_back()`, which works like `.append`.
- `.at()`, which works like the normal index operator `[]`, but which checks for out-of-bounds reads (it *throws an exception* in this case – which is far better than the code silently running incorrectly).

L18x4_vect.cpp

Actual Vectors

To explain a couple more methods, we need to discuss *iterators*. Iterators are objects, whose main data member is simply a pointer to an element of a `vector`.

For a given `vector` named `x`, then, for example,

`x.begin()` would be an iterator pointing to the first element of the vector;
`x.begin() + 2` would be an iterator pointing to the third element of `x`;
`x.end()` would be an iterator pointing to one entry past the end of the vector.

In all cases, you would use `*` in front to access the value of the element itself.

You can also set iterator variables using a declaration like

```
auto it = x.begin();
```

where `auto` is a declaration for variables whose type can be deduced from its value.

You can see how to use these in loops in **L18x5_iter.cpp** .

Actual Vectors

Iterators can also be used in the following methods:

- `x.erase(it1, it2);` will erase the elements from `it1`, up to, but not including, `it2`. Using a single iterator will simply erase a single element.
- `x.insert(it, value);` will insert `value` into the array at the element that `it` is pointing to, pushing everything else back one position.
- `x.insert(it, other_it1, other_it2 );` will take all the elements between iterators `other_it1` and `other_it2` for some other vector, and insert them in at iterators `it` in vector `x`.

L18x6_middle.cpp

A quick warning about iterators: if you perform insertions or erasures on a vector, existing iterators on that vector can get *invalidated*: they may no longer point to the element that you expect them. Don't use an iterator to traverse a vector if the vector changes size as you iterate.