MTH 4300, Lecture 17

The Implementation; Destructors; `operator=`

E Fink

April 2, 2025

1. The Implementation

Here is the basic class definition, which may need to be filled out later:

```
class List {
private:
    string *listptr;
    int capacity;
    int num_entries;
    void reserve();
public:
    List(int);
    List();
    void append(const string&);
    string& operator[](int);
};
```

L17x1_list

The Implementation

`listptr` points to the dynamically allocated array with length `capacity`, which contains the contents of the List. `num_entries`, as mentioned before, is the number of entries currently being used; this number should always be less than or equal to `capacity`.

Methods:

The constructor `List(int)` will create a `List` with `capacity` given by the argument, but with no entries initially.

Since we have made a constructor with parameters, C++ won't provide an automatic default (no parameter) constructor, so we may want to put one in ourselves. This would get called if, for example, we declared an array of `Lists`.

The Implementation

The private `reserve()` method performs the growth process we described: an array twice the current capacity will be allocated, entries from the old array will be copied into the new array, the old array will be deallocated, and `listptr` will be assigned to the new array.

The method `append(string)` first checks if there is room left in `*listptr`: i.e., if `num_entries < capacity`. If not, the function `reserve()` is called, which creates an updated list for `listptr` to point to. Either way, the method then proceeds to add the argument as the next element in the list, which will be the entry with index `num_entries`, and then `num_entries` will be incremented by one.

The Implementation

Finally, `operator[]` allows us to use indexing to retrieve elements from the list. This method returns a reference to the entry itself: that way, there will be no unnecessary copying of the return value, and we can use bracket for reading AND writing.

This last part is worth focusing on. With most functions, member or non-member, code like

`3 + 5 = x;` or `f(5) = y;`

are illegal, because `3 + 5` and `f(5)` have their return values typically stored in temporary, read-only locations in memory (often not in stack or heap).

On the other hand, if `mylist` is a `List` object, then when

```
mylist[2] = "Hello";
```

evaluates, since `mylist[2]` returns an alias for an array element – when the return value is a REFERENCE! – the string literal `"Hello"` can be stored there.

2. Destructors

Variables and objects have *lifetimes*. Variables created on the stack have *automatic lifetime*: they exist until your program reaches the end of the scope they were declared in (the closing curly brace, typically). Objects created on the heap, using `new`, survive until they are explicitly `deleted`.

In either case, all variables and objects have an end-of-life. When any object reaches the end of its lifetime, a special function called a *destructor* gets called.

We haven't noticed it so far because

- it gets called automatically
- it gets written automatically by the compiler if you don't do so
- for our classes so far, the destructor doesn't do anything.

Destructors

However, consider the code in **leak.cpp**. In the current state, this code suffers from memory leaks.

In the code, the variable `x` gets created and destroyed over and over again. Each time through the loop, `x` gets reconstructed, and its three attribute variables get added and removed from the stack – and some memory gets reserved.

But the reserved memory never gets released! Examine the code: none of the functions that get called from this code ever call `delete[]`.

Destructors

Fortunately, there is a solution. You can write your own destructor, which still gets called at end-of-life – and typically, this function's purpose is to release resources (like heap-allocated objects) when you are done with them.

To declare a destructor: `~ClassName();`

To implement a destructor:

```
ClassName::~ClassName(){
   // some deletes?
}
```

(Try building `leak.cpp` with and without the one-line body of the destructor commented out. When it is commented out, so that there is no working destructor, it is slow/crashes.)

3. operator=

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

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

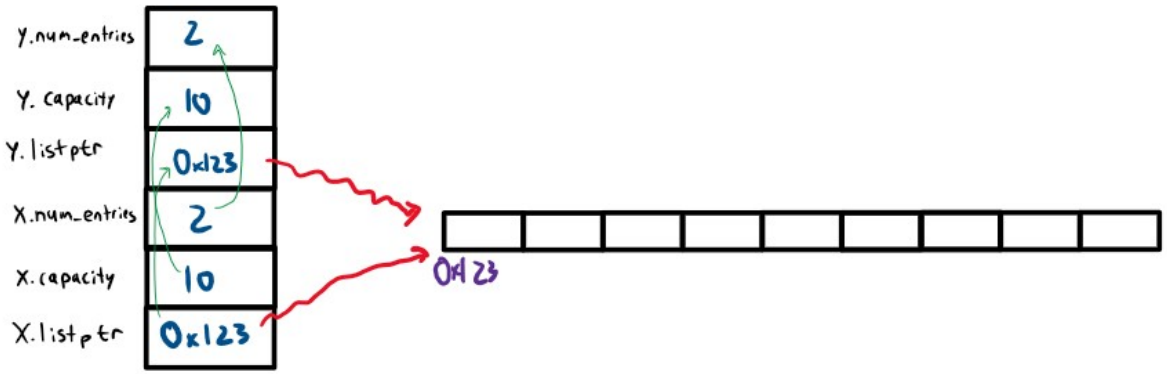
What happens in the line `y = x`?

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.)