MTH 4300, Lecture 16

Separate Compilation; A Bit about the Build Process; A List Class; The Implementation

E Fink

April 2, 2025

1. Separate Compilation

For a large project in any language, it is customary to separate code into various files. This makes it easier to organize projects, locate code, and build incrementally. It also allows individual files, representing incomplete programs, to be compiled separately – so that if only one file needs to be changed, only one file needs to be rebuilt.

A very common strategy is to separate class definitions from client code. In fact, we can go a little further, and put

- class DECLARATION in one *header* file, typically with a `.h` extension;
- class IMPLEMENTATION (e.g. method definitions) in another, `.cpp` file;
- and client code, including `main()`, in a different `.cpp` file.

Separate Compilation

L16x1_complex_folder

How does one actually compile and run this? Here's a VS Code-oriented route.

1. Open up VS Code, and select "Open Folder", choosing the folder that contains all your files. Make sure everything is in the same folder.
2. Also, make sure that that the two `.cpp` files `#include "complex.h"`, and that `complex.h` itself has *include guards*:
   `#ifndef COMPLEX_H`
   `#define COMPLEX_H`
   and `#endif` at the end.
3. Open up a terminal in the directory containing your code, by selection "New Terminal" in VS Code.
4. For the GCC compiler, type `g++ -c complex.cpp` and press Enter. Then type `g++ -c client.cpp` and press Enter.
5. Next, type `g++ complex.o client.o -o finalprogramname`
6. And then run, with `./finalprogramname`

Separate Compilation

Notes:

- If a constructor or other method has **default parameter values**, these default values should appear in the *header* file and NOT in the implementation file.
- If a constructor has an **initializer list**, that list should appear in the *implementation* file and NOT in the header file.
- Just like other methods, the **body of the constructor** (which is sometimes just {}) should appear in the *implementation* file, not the header.
- Make sure your method signature lines in the implementation match their declarations exactly (aside from parameter names), including `consts` and `&`s!

2. A Bit About the Build Process

Why is all this necessary? (And why must the class declaration and implementation be separated?) Well, secretly, what we've been calling "compilation" is actually several different things: among other things, there is preprocessing, linking, and compilation.

Appropriately enough, preprocessing comes first. All those lines that start with # are preprocessor directives. These insert the contents of files into your `.cpp` files. (Lines like `#include <iostream>`, where the header to be included is enclosed in <>, do the same, except they insert standard library headers, rather than user-defined ones.)

The `#ifndef COMPLEX_H`, `#define COMPLEX_H`, and `#endif` preprocessor lines ensure that the contents of `complex.h` only get inserted once into a given file – the compiler doesn't like it if you put the same declaration twice.

A Bit About the Build Process

After the preprocesser has done its job, then comes actual compilation (well, to be precise, compilation and assembly), which produces so-called *object files*.

Look at `complex.h` together with `client.cpp`. Every variable, function, and class referenced in the combined file will at least have been DECLARED, even if not provided with a DEFINITION. So `client.cpp` is ready to be compiled: compilation will produce an almost-ready-to-run program, with a few references to functions whose definitions will be supplied later. This almost-complete program is what an object file is.

You create this object file by `g++ -c client.cpp`, and the object file will be called `client.o`.

Likewise, the class implementation file `complex.cpp`, which includes `complex.h` as well, can also be compiled to an object file – this object file is not-quite-complete because it lacks a `main()` function!

A Bit About the Build Process

Finally, the job of the *linker* is to take the object files, and connect the missing definitions: this is what the line `g++ complex.o client.o -o blah` does. (The `-o blah` sets the name of the output executable to `blah`.) The linking will be facilitated by the fact that `client.o` and `complex.o` each were built using the same declarations in `complex.h`.

For all this to work, C++ requires there to be:

- ONE `main()` function;
- ONE **definition** IN TOTAL of each function, class, and globally declared variable;
- and ONE **declaration** PER FILE of each variable, function and class used. [More accurately, one declaration per translation unit – a file together with its includes.]

These demands are why class declarations and implementations are separated: they allow the class implementation to be recompiled separately, periodically, without running afoul of these rules.

3. A List Class

We'd like to create a class that acts a bit like Python's `list`, for `strings`. It should support things like appending and erasing.

Using plain arrays won't work so well – they are fixed in size, and that size needs to be known at compile time.

So we will write a class whose primary member variable will be a dynamically-allocated array. The class will also contain some extra member variables; and most importantly, it will contain fancy `list`-like methods (well, the primary one we will implement is `append`, which is the most interesting one anyway).

(BTW: `vectors` do what we are describing. I'll finally cover these shortly, but first we'll make our own miniature version, to see how they work.)

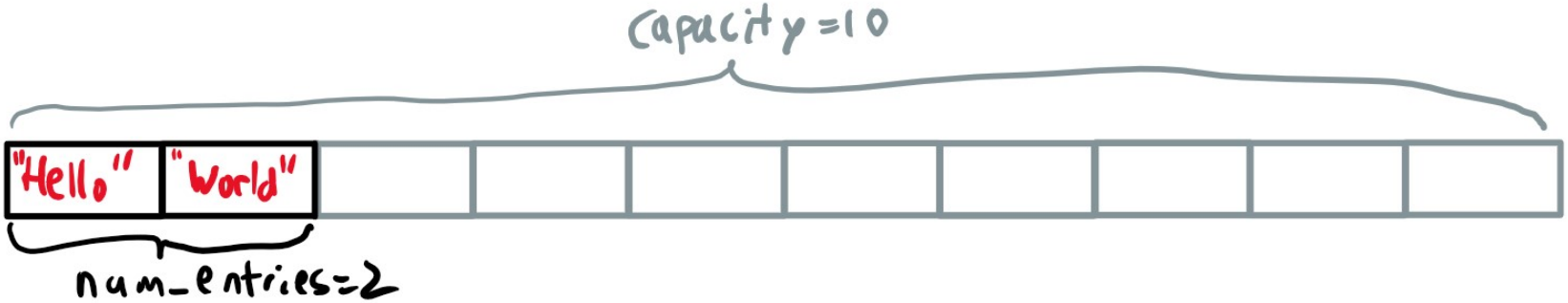
A List Class

Here's an outline of how our `Lists` will work. They will have TWO lengths: the length of the allocated array, and the number of entries currently be used. Huh?

Here's what I mean: imagine that I write

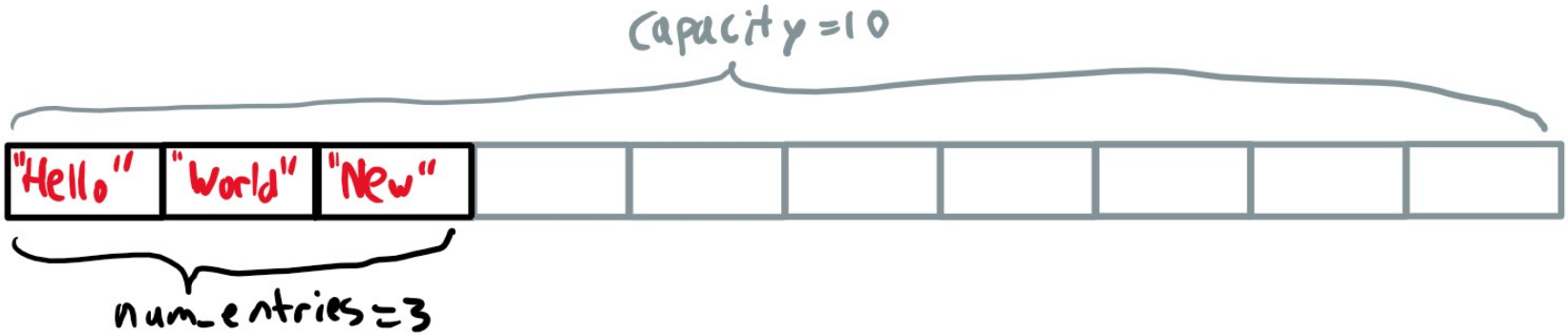
```
List mylist(10);
mylist.append("Hello");
mylist.append("World");
```

In that case, `mylist` should be thought of as a list that contains two elements, `"Hello"` and `"World"`. Behind the scenes, there will be a dynamically allocated array of length 10; but a member variable called `num_entries` will be set to 2. That means that effectively, we will pretend that the other 8 entries aren't there ...



A List Class

. . . until someone tries to `append` an element: at that point, the element will be placed in the third entry, and `num_entries` will be updated to be 3.



After a few more appends, all ten entries will be filled in, and `num_entries` will be at 10. If we call a subsequent `append`, what happens next?

Answer: at this time, before doing the append, we will call a function called `reserve()`, which will:

- create a new dynamically-allocated array with twice the length of the previous one;
- copy all elements of the old array into the new one;
- `delete[]` the old array, and then proceed.

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);
};
```

L16x2_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.