COMP 161 - Lecture Notes - 04 - The Structure of a C++ Program

January 15, 2015

We begin our exploration of C++ by considering the high-level structure of a basic C++ program in the style that we'll be developing in this course.

Procedures and Statements

In COMP160, your Racket programs were a collection of definitions, mostly functions, and this is true of the first kind of C++ program we'll be developing in this course. There are several key differences we need to be aware of though.

Racket functions were PURE FUNCTIONS. They took inputs, produced an output, and had no side effects. In C++ our "functions" have the option of taking no input, producing no output, and causing side effects. They can interact with program STATE VARIABLES or system I/O DEVICES. For this reason we'll use a more general term[1]: **Procedure**.

[1] as opposed to function, which is a well defined object from Mathematics

Racket functions were constructed as a series of nested EXPRESSIONS. An EXPRESSION has a specific value associated with it and so we can always SUBSTITUTE the expression for its value[2]. Order of execution was determined by nesting; the inner most expressions were executed first. In C++, a procedure is written as a sequence of STATEMENTS. STATEMENTS are primarily written for effect, not for value[3], and more or less correspond to a command issued to the computer[4]. The order in which STATEMENTS within a PROCEDURE are executed corresponds to the order[5] they appear on the page. In short, a C++ procedure is a sequence of imperative statements issued to the computer.

[2] this is how the Racket interpreter worked. Go use DrRacket's stepper to see it in action

[3] However, they are typically built out of at least a few EXPRESSIONS

[4] aka an IMPERATIVE

[5] top to bottom

The style of Racket programs we wrote in COMP160 revolved around the use of pure functions and so it's called FUNCTIONAL PROGRAMMING. The style of C++ programming we'll begin with revolves around procedures and imperatives and so we call it *Imperative, Procedural Programming.*

Main

Every C++ program must have one and only one procedure named *main*. The main procedure is, in effect, the program; when you compile your program, the executable that results executes the main procedure of your program. We did not have such a strict requirement in Racket[6]. However, you may have encountered this style when dealing

[6] or bash

with Universe programs in BSL Racket where it's typical to write a main function which invokes the big-bang function.

The main procedure requirement can and will cause a few headaches:

1. Attempting to compile to an executable without a main procedure results in a long, seeminly cryptic sequence of errors.
2. We'd like to compile and run some tests separate from our main program. Running tests[7] requires a main. So most of the time we'll need at least two separate programs, and therefore two separate files: one for tests and our actual program.
3. In order to test code it must be compiled along with the main that runs the tests. In order to include it in our main program, it must be compiled with our main procedure. This forces us to put any code we want to test in a file separate from our program's main procedure and our test's main procedure[8]

[7] any code

[8] combining test logic and main program logic quickly becomes a bad idea

The result here is that organized, well-tested code requires multiple files and will require several different compilation procedures. Thankfully, we have some tools that will help expedite the process of compiling and building our programs. So when compiling becomes time consuming, we can turn to these tools.

Libraries

We'll strive to section off a large chunk of our program code into files separate from the program's main procedure. Essentially, we want to build LIBRARIES of code that can then be used by the main procedure[9]. Libraries are a vital part of software development for two reasons:

[9] This is analogous to the teachpacks you're used to from COMP160

- They allow us to more easily test our code outside of the normal, expected execution of the program by separating code from the main procedure.
- They allow the library code to be reused in other programs without copying and pasting from one program to the next.

The C++ libraries we'll be developing make use of a two file file format: a HEADER FILE and the IMPLEMENTATION FILE. The HEADER FILE contains documentation and declarations of procedures[10]. It tells you and the computer what's in the library and how it may be used, but does not tell you or the computer how the library code actually works. The IMPLEMENTATION FILE contains the complete definition of the library; it tells you and the computer how the library does what it does. Another way to look at it is that the

[10] and eventually other things like structures

HEADER simply provides a description of the *interface* provided by the library. By physically separating the library's interface from it's actual implementation we give ourselves the chance to swap in different[11] implementations later on. The power and impact of this idea cannot be overstated.

[11] presumably better

A more immediate and practical result of this two file style is that once you've created a header, you can produce code that can be compiled to an intermediate, non-executable stage called object files. Using the header only, the compiler can at least recognize that library procedures are being called more or less correctly. So compiling to object files gives us an opportunity to quickly and easily fix typos and syntax errors prior to debugging logic errors. If you stick to this regime, then you'll end up debugging in lots of small chunks rather than one giant debug session. The former tends to be much less frustrating than the later.

C++ namespaces

The more we draw on libraries of C++ code, the more likely it is that we'll need or want to name one of our procedures the same thing as a procedure form another library. If the SIGNATURE[12] for our procedure is different from existing definitions, then it's not a big deal. We can simply OVERLOAD[13] the procedure name. However, it may very well be the case that we'd like to provide a new definition for an existing name and signature. We could OVERWRITE[14] the other definition, but this is generally frowned upon[15]. Sometimes we actually want both definitions available as they could each be the best choice in different scenarios. For example, one might work best for small sets of data where the other works best for large sets. In this case, we need a way to differentiate the two definitions of the procedure. For this we use namespaces.

[12] number and types of inputs and type of output

[13] provide multiple definitions with different signatures

[14] replace an existing definition

[15] until we get to Object-Oriented Programming

The C++ namespace is a way to logically group definitions within a named space. Last names, or family names, act as a type of namespace for people. My wife and my sister-in-law are both named Sarah. If I'm talking about my wife, I can say "Sarah Mayfield" and remove any ambiguity. If I need to talk about my sister-in-law, I can talk about "Sara Wells"[16]. So, namespaces allow us to effectively attach secondary names to procedures. In fact, you can put names spaces in namespaces just like we can have middle names or even numbers. I'm actually James Logan Mayfield IV. My son is James Logan Mayfield V. The numeral part of our names can act as a namespace to differentiate the two of us. Once we put C++ definitions within a namespace, then we can always attach the namespace specifier to that name when we use it in order to unambiguously refer to a definition.

[16] Ignore the different spelling. We're talking about verbal communication

You typically don't call people by their complete names. Instead, we let context differentiate people more often than not. If I'm at home and I say Sarah, then the default assumption is that I'm working in the Mayfield namespace and I'm talking about my wife. We can do the same thing with C++. A *using namepspace* statement can be used to declare a default namespace. As many of the build-in C++ libraries use the standard[17] namespace, we'll often make use of the *using namespace std;* statement in our procedures that utilize those libraries. This saves us from having to call library code by it's "full name".

[17] or *std*

When definitions aren't placed in a namespace, then they go to the *global* namespace. It's often considered bad practice to put definitions in the global namespace. Over the years software engineers have learned that it tends to cause problems as programs grow. We could probably get away with it in this class as the size of our programs are small by industry standards. However, let's start out with good habits and not ignore industry best practices. We'll always declare our library code within a namespace.

Unit Tests

The style of testing you learned in COMP160 is called Unit Testing. In unit testing you write lots of little tests for the small parts[18] of your program. Our units are procedures, so this means we want to test each procedure just like we tested every function in Racket. Unlike in Racket, we'll be putting all our tests in a separate file from the code it's testing. The reason is highly practical: we don't want to include the compiled test code with the finished product. Our users should not need to rerun our tests[19]. Furthermore, the compiled tests will cause the size of the executable to go up. The tests are for us, the developer, not the user. So, we put them somewhere else so that we can exclude them from the finally, "shipped" product.

[18] units

[19] nor will they probably want to

The File Struture of a C++ Program

Hopefully you're starting to get a clear picture of the the different files used for organizing our C++ programs. Just to be certain, let's recap what we've talked about. Minimally we're looking at:

1. One file containing the program's *main procedure* definition.
2. Three files per library: One header, one implementation file, and one file containing unit tests for the library procedures.

Libraries typically group procedures by logical purpose. In addition to avoiding name conflicts with other libraries, we'll often use

namespaces for fine-grained, logical grouping within a library. It's not unlikely that we'll want to write several libraries for one project. This means that the number of files in or project can quickly grow into the double digits. We have lots of tools to help us manage this complexity, and so we'll embrace this organizational style because it leads to reusable code that is easier to test and maintain[20].

[20] A key goal in software engineering

Even though all of our files contain C++ code, we use two different file extensions: one for header files and one for all other files. Our library header files should all have the file extension **h**. All other C++ files have the extension **cpp**. For example, in class and lab we'll look at a program containing the following files: factorial.h, factorial.cpp, fact_tests.cpp, lab3_main.cpp. The first two files make up the factorial library which is tested in the third file. The final file contains the main procedure for our program. Now that we know how and why we spread our code across multiple files, we need to look at how we can use and direct the compiler to glue it all together into a single executable and how to manage this as our usage of libraries

Library Boilerplate

All the libraries you'll write have some basic boilerplate code that goes in them.[21]. Much if this boilerplate is used to connect headers to implementations and libraries to the code that uses them.

[21] You can find all this code in the factorial library used in lab 2 and discussed further in lecture notes 4

There is one thing we put in every file that is not C++ code: a file header[22]. File headers are comment blocks meant to inform the human reader. Minimally, your headers should contain two pieces of information: the name of the author, and a brief description of the file contents and purpose. Leaving the file header out won't break your code. It's just bad style. Remember, code is read by humans first and computers second. These headers are meant to give programs some direction as to the contents and purpose of the library so that they can better read and interpret the C++ code.

[22] different than a header file!

The library header file (the .h file) always contains two key pieces of code: define guards and a namespace declaration. The define guard has three parts: #ifndef , #define , and #endif . We read the # as "pound". The *ifndef* as "if n def", which is short for "if not defined". The rest are fairly self explanatory. We'll need to #include our library header file in multiple source files[23] and without these guards, the compiler will think we're trying to re-define some procedures when it encounter a repeated include and complain about multiple definitions. The namespace declaration places our procedure declarations within a namespace. Notice that the define guards surround all the library code and that function declarations will go within the namespace block delimited by the curly braces; your pro-

[23]

1. Library Implementation
2. Library Tests
3. Program main file

cedure declarations go within these two code elements.

To connect the header declarations with the implementation definitions, the library implementation file *must* include the library header with a #include[24] statement at the top of the file, before the actual definitions. By doing this, we've directed the compiler to glue together the what and how of our code. This is important because to use a library in other files we only include the header, the "what" part of the library. This allows the compiler to recognize correct usage of library code, but doesn't include the details for executing correctly used code. The inclusion of the header with the implementation ensures that when the complete program compilation is carried out we'll know the exact definition for all the library procedures.

[24] "pound include"

The test files uses library procedures and code from the gtest library. So, we need to #include both out library header and the gtest library header at the top of this file. Generally speaking, you should write a test case for every procedure. The one exception to this rule might be combining tests for helper procedures with the tests for the procedure their helping. In this case, the best thing to do might be to write one test case for the whole thing, but split the tests out by procedure[25]. We'll talk more about testing soon.

[25] you see this in the factorial library from our last lab

You're going to write a lot of libraries, so get used to this code. An industrious student might take a crack at writing a bash script to automatically generate starter files for libraries. Such a script might take the library name as an input and then produce these three files with the boilerplate all filled in. Or you can do a lot of typing, copying, and pasting at the start of every program you write. It's your time so its up to you.