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

January 14, 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.

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[12] 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[13]. 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.

[12] units

[13] nor will they probably want to

File Purpose and Types and usage

You should be getting a more clear picture of the different files used for organizing our C++ programs. 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. So one project can have many libraries and therefore many files. Again, we have lots of tools to help us manage this complexity, so we embrace this organization style because it leads reusable code that is easier to test and maintain[14].

[14] A key goal in software engineering

Library HEADERS 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