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

Spring 2014

We begin our exploration of C++ by consider the high-level structure of the basic C++ programs well spend the first part of the semester developing.

Procedures

In COMP160, your Racket programs were a collection of 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. 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++, procedures are 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.

[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 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 or bash.

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 cryptic sequence of errors.

2. We'd like to compile and run some tests separate from our main program. Running tests[6] 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.

[6] any code

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[7]

[7] combining test logic and main program logic quickly become 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[8]. Libraries are a vital part of software development for two reasons:

[8] 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 resorting to copy-paste jobs.

In C++, libraries often involve the use of two files: a HEADER FILE and the IMPLEMENTATION FILE. The HEADER FILE contains documentation and declarations of procedures[9]. 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 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[10] implementations later on. The power and impact of this idea cannot be overstated.

[9] and eventually other things like structures

[10] presumably better

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[11] of your program. In C++, 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[12]. 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.

[11] units

[12] nor will they probably want to

File Types and usage

It's clear now that the typical C++ program is split across several files:

1. A file containing the program's *main procedure* definition.
2. Two files per library. One header and one implementation file.
3. A file containing unit tests for the library.

Library HEADERS will 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. The important thing at this point is understanding how we glue all this code together.

Include Directives and Linkers

When you're writing C++ that makes use of a library[13], then you typically use an INCLUDE DIRECTIVE in the file that uses the library code. Include directives direct the compiler to *include a library header*[14] with the current file. Note that by including headers, we lack the actual implementations of the library code we're using. To solve this problem we turn to the compiler, or more specifically the Linker. As we'll see in our next set of notes, the linker is the part of the compiler that is capable of stitching together code from different files and and thereby providing the missing implementations of our library code.

[13] your own or one from a third party

[14] always include .h files, never .cpp