COMP 161 - Lecture Notes - 05 - The Compiler

Spring 2014

In this class we explore the compilation process and the basic usage of our compiler, the GNU Compiler Collection's[1] C++ compiler **g++**.

[1] GCC

Our Example Program

We'll be exploring the compilation of a program consisting of four files:

1. *factorial.h*
   The HEADER FILE for the factorial library.
2. *factorial.cpp*
   The IMPLEMENTATION FILE for the factorial library.
3. *fact_test.cpp*
   The UNIT TESTS for the factorial library
4. *lab3_main.cpp*
   The MAIN PROCEDURE that utilizes the factorial library

Our goal is to understand how to build and test our library as well as the main program. In doing so we'll also look at some other ways in which the compiler assists us with the correctness and efficiency of our code.

Basic Compilation

The default behavior of **g++** is to take one or more cpp files[2] and turn them into an executable program named *a.out*. The following command will attempt to carry this process out.

[2] you can compile h files. but we won't in this class

```
g++ factorial.cpp lab3_main.cpp
```

One of two things will happen when we run this command. Either the compiler will find errors in our code, report those errors, and fail to create the executable, or it will create the executable and produce no output on the standard output. So, if it looks like nothing happened, then chances are everything went OK. Once we have the executable, then we can run our program just like any other CLI command[3]:

[3] the ./ is likely necessary if the directory containing your program isn't in your PATH variable

```
./a.out
```

We'll pretty much never use this form of the **g++** command. First off, the name *a.out* is terrible and if we're building multiple programs we can't have them all with the same name. More importantly, we want the compiler to be a bit more particular about our code and not just tell us when it's totally wrong but warn us when we do something that may lead to problems[4]. For the first issue we'll use the **-o** ***output-name*** option to specify the name of the output. For the second, we'll use **-Wall** to get compiler warnings as well as errors[5]. Notice that we're bucking convention a bit here and putting the optional arguments after the required arguments[6]

[4] Correctness >> convenience

[5] `http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`

[6] the file path in this case

```
g++ factorial.cpp lab3_main.cpp -Wall -o factorial
```

This new command produces the well-named factorial executable and will, if needed, warn us when our code looks problematic. You may use this command on occasion, but typically we'll do a multi-stage compilation that makes a pit stop on the way to the executable.

The Compilation Process

Our compiler carries out a four stage process[7]:

[7] compile with the -v option to see the commands run by the compiler

1. Preprocessor
2. Compiler
3. Assembler
4. Linker

The g++ compiler has options which allow you to control how much of this process is carried out[8]. Absent these options, it will take whatever you give it and attempt to finish the compilation process.

[8] `http://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html`

Preprocessor

The preprocessor essentially takes our C++ and writes some other C++ from it. Most notably, this stage takes care of all the statements beginning with **#**. The most notable of which is *#include*, which tells the preprocessor to copy and paste a header file in to a file. The preprocessor is also a vital component of our unit testing framework, gTest[9]. The tests we write look like procedures, but are in fact Macros[10].

[9] `https://code.google.com/p/googletest/wiki/Primer`

[10] A macro gets expanded into C++ code by the processor

The option **-E** causes the compiler to stop after preprocessing and then print the output to the terminal. We can use *-o* to name the output[11]. To see the what happens to the include directives, we could do.

[11] or > to redirect to a file

```
g++ factorial.cpp -E -o pp_factorial.cpp
```

For a really drastic transformation, look at what happens to our unit test macros:

```
g++ fact_tests.cpp -E pp_fact_tests.cpp
```

Compiler

The compiler is where the whole system gets it name. It turns high-level C++ into low level assembly. We can stop the compiler after it compiles using the **-S** option. By default, this option produces an assembly code file with the *s* extension but the same name as the file being compiled[12]. If you want to see some assembly try:

[12] so we don't need to name the output

```
g++ factorial.cpp -S
```

To view the assembly code, just open the newly created *factorial.s* with emacs or less. The assembly code is the most accurate representation of what the computer really does when our program is running. When you're doing supper fine-tuned optimizations or tracking down really nasty bugs, you might have to look at the assembly. Thankfully, we'll never be in this situation in this class.

Assembler

The assembler takes human readable assembly[13] and produces a machine readable OBJECT FILE. The **-c** option will produce an object file with the same name as your source file[14] but with the *o* extension. This stage is as far as you can go without a main procedure and so *we'll compile to object files often.* Use the following command to build an object file for the library.

[13] assuming you know the language

[14] cpp file

```
g++ factorial.cpp -c
```

You can open the resultant *factorial.o* with emacs and clearly see that, to the human eye, it's gibberish.

Linker

The linker stitches together object files and creates an executable. This means that one[15] of the object files must contain a main procedure. There's no special command to link, just don't use any of the other options. So, assuming we used the option *-c* to create *factorial.o* and *lab3_main.o*, we could build our executable as follows:

[15] and only one!

```
g++ factorial.o lab3_main.o -o factorial
```

We're back to default behavior where executables named *a.out* are produces, so we'll use the option *-o* to specify the name of the output.

Back to Basics

Let's return to our basic command:

```
g++ factorial.cpp lab3_main.cpp -Wall -o factorial
```

We like this because it's a one-liner, but there's two real problems with this approach:

1. If we haven't compiled yet and these files have errors, then we're stuck with errors from two files. It's probably better for our sanity to work on and fix one file at a time.

2. We have to compile the factorial library for its tests as well. If we compile it to an object, then we don't have to recompile it so much.[16].

So, a multi-stage compilation where individual files are compiled is probably in order. First, we'll compile each cpp file to an object:

```
g++ factorial.cpp -Wall -c
g++ lab3_main.cp -Wall -c
```

Then, we link them:

```
g++ factorial.o lab3_main.o -o factorial
```

As you'll see, it's not likely that you'll need to run all of these commands every time you compile. Eventually, we can use the program *make* to expedite the multi-stage compilation process.

As we proceed, there's one important thing to keep in mind:

If you change a cpp file you need to recompile the object file associated with that file.

It's easy to forget to do this. I often see students fix a bug in factorial.cpp and then just recompile the main executable with the link command. Doing this typically reuses the object corresponding to the old buggy code. At that point, one questions their sanity as they're certain the changes they made should make something different.

[16] We'll work with small libraries, but in industry, re-compiling might mean minutes or hours of down time while the compiler works

Building and Running Tests

So far we've been looking at compiling our main program that utilizes the library and the main produce. Before you even get that point you should be compiling and running your unit tests to verify that your library code is working as intended. Building our test program requires some more involved compilation options, so we put it off until after the basics.

Before we look at the compiling commands, I want to point out two key elements of using the gTest framework:

- The gTest framework provides a ready made main procedure for running unit tests. Rather than write our own, we can link to the library containing their main when we compile. This is good, because it lets us focus on the testing logic of our code, and not particulars of running gTest tests.
- The gTest framework requires us to not only use include directives in our code, but to explicitly link to several libraries with the g++ linker.

First things first, we need to compile our library and our unit tests down to object files. This can be done without any special compiler options.

```
g++ factorial.cpp -Wall -c
g++ fact_tests.cpp -Wall -c
```

This is good! It means we can still quickly compile these files and check for compiler errors and warnings.

Once we get clean object files, we need to link them together and link them to three libraries:

1. *pthread*
   The Posix Thread library. It's used by gTests. To link, use the g++ option *-lpthread*
2. *gtest*
   The main gtest library. To link, use the g++ option *-lgtest*
3. *gtest_main*
   The gtest main procedure. To link, use the g++ option *-lgtest_main*

The command to link all these libraries together and build an executable called *fact_tests*looks like this:

```
g++ factorial.o fact_tests.o -lpthread -lgtest -lgtest_main -o fact_tests
```

Notice we didn't used *-Wall*. Presumably we cleared out all those problems when we compiled our objects, so adding that option now isn't necessary. Further more, -Wall is really a compiler option and this command should only use the linker[17].

The last thing I need to point out is that the order of arguments matters here. If you run this command[18]:

[17] objects and link options only

[18] your objects before the link options

```
g++  -lpthread -lgtest -lgtest_main -o fact_tests factorial.o fact_tests.o
```

You'll get a long list of errors. So, as a rule of thumb, *write compiler commands in the opposite style of other commands, namely* CMD ARGS [OPTIONS].

Running your Tests

The pre-built main we're using to run our tests will, by default, run and report on every test your write. With the use of some commandline arguments, we can control which tests it runs[19]. This is good! You're typically fixing one procedure at a time, so running tests for other procedures is a distraction.

[19] `https://code.google.com/p/googletest/wiki/AdvancedGuide#Selecting_Tests`

First we need to notice how tests are identified. When we write a test we specify a test case and test name as follows:

```
TEST(case,name){
  //test code
}
```

In *fact_tests.cpp* we have four cases: *ver1*, *ver2*, *ver3*, and *ver4*. The second and third versions have two named tests: *factorial* and *factHelper*. Lets proceed as if our test program were named *fact_tests*. To see a list of all the tests cases and names we'd run:

```
./fact_tests --gtest_list_tests
```

To run the test factorial from case ver2, we'd then run:

```
./fact_tests --gtest_filter=ver2.factorial
```

To run all of the tests in ver2 we can do:

```
./fact_tests --gtest_filter=ver2.*
```

As you can see, we can use the * wildcard in forming the filter string. Odds are we either want to run one test or a whole test case, so the above examples pretty much cover our use cases.

Compiler Optimizations

So, your tests pass and your program is, as far as you can tell, working as intended. At this point we can unleash the compiler and let it attempt to make our code faster[20]. Modern compilers are able to carry out common optimizations that can sometimes really boost the performance of our program. The g++ compiler carries out three levels of standard optimizations. The higher the level number the more optimizations done by the compiler. The options **-O1**, **-O2**, and **-O3** turn on the different optimizations. They should be used to compile objects or executables from source code[21]. The following demonstrate the use of compiler optimizations.

[20] `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

[21] They're done by the compiler and so cannot be carried out at the link stage of the process

```
g++ factorial.cpp -c -Wall -O2
g++ lab3_main.cpp -c -Wall -O2
g++ factorial.o lab3_main.o -o factorial
```

Notice that separate compilation gives the the chance to do different optimizations on different sets of code.

Compiler optimizations effectively rewrite our code such that the assembly produced by the compiler might not exactly match up with the C++ we wrote. It's possible that the behavior of optimized program might differ from that of the C++ code we wrote. This can make debugging difficult unless you get the optimized assembly and then debug the assembly! We'll typically throw in compiler optimizations after we've fully tested our code and have turned our attention to program efficiency.[22].

[22] Don't optimize until you're confident the program is correct

Compiling for Debugging and Profiling

When we encounter tricky logic errors and runtime bugs in our code we often turn to debuggers like the program GDB or memory system checkers like Valgrinds memcheck. When compiler optimizations don't seem to cut the mustard and we have to optimize our programs by hand, then we turn to profiling tools like the Valgrind suite. Both programs require some special annotations be added to our code so that they can better communicate their results to us. The compiler can take care of this for us with the **-g** option[23]. This option should be used when object files are compiled and when they are linked. The following sequence of commands produces an executable named *factorial* that is suitable for debuggers and profilers.

[23] `http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html`

```
g++ factorial.cpp -c -Wall -g
g++ lab3_main.cpp -c -Wall -g
g++ factorial.o lab3_main.o -g -o factorial
```

We can also use the debugger option with our tests. When we're optimizing our code with the help of profilers, we might throw in compiler optimizations as well. It's generally safer and easier to leave them out and then toss them back in after we've finished our own optimization process.

Putting It all together: A Work flow

So we've put our program through the paces and have come up with what we think is a solid design for our program. Now it's time to code. Compilation is a vital part of this process and in this class we'll consider a work flow similar to this:

1. Fill in stubs[24] for our library functions and compile the library to an object to check for basic syntax errors and compiler warnings. Correct errors and warnings as needed.

[24] like templates

2. Write unit tests for the library and compile the tests to an object. Correct all syntax errors and warnings found in the tests.
3. For each procedure in the library:
   (a) Complete the procedure and compile the library to an object. Correct syntax errors and compiler warnings.
   (b) Link library object and tests and run tests.
   (c) If tests fail then recheck test and code logic and fix the bug. Recompile objects as needed and relink to get an updated test program. Rerun tests and repeat debugging as needed.
   (d) If debugging proves difficult we might recompile with debugging turned on and run our code through *GDB* or Valgrind's *memcheck*.
4. Stub the main procedure. Compile to check for syntax errors and warnings. Correct as needed.
5. Complete the main procedure, incrementally compiling, running, and testing the program.
6. When the main procedure is complete, recompile with optimizations.
7. If further optimization is needed, recompile for profiling and utilize tools like Valgrind's *cachegrind* to seek out bottlenecks in memory performance and Valgrind's *callgrind* for bottlenecks in CPU performance.

If you examine this work flow closely, you'll see that several different compilation procedures are needed and these procedures require the use of various options. The most used options involved in these procedures were all discussed above and are listed again in table 1. You should also note that regular compilation of code is used in an effort to reduce the number of compiler errors and warnings we have to deal with at one time[25]. Compile early and compile often.

[25] Easier to correct one procedure than it is to correct ten

Option	Description
-o *filename*	provide a name for the file output by the compiler
-Wall	provide warnings in addition to errors
-c	Stop before linking and produce an object file
-l*libname* i.e. -lpthread -lgtest -lgtest_main	Link with library *libname*
-O{1,2,3}	Compiler optimization level 1, 2, or 3
-g	Annotate for Debugging

Table 1: Key options for g++