Useful Programming Tools

Pete Willemsen <willemsn@d.umn.edu>, Last Updated - January 21, 2014

There are several tools that will be useful for you when programming. I would like you to become familiar with two of these tools: the GNU Debugger *gdb* and the CMake cross platform build system. Debuggers are very useful and if you spend a little time learning how to use them, whether it is gdb or Visual Studio's debugger, it can save you time and make you more efficient at coding. CMake is similar in that it can make you more efficient at building programs that are cross-platform from the beginning. In other words, you can write code that will run on Linux, OS X, and Windows without having to invest much time trying to figure out how to make your program build on these systems.

GDB - GNU Debugger

The GNU Debugger, *gdb*, lets you inspect your running programs as well as your programs that are failing due to segmentation faults, bus errors, or other reasons. However, to use gdb, you need to compile your programs to include debugging information. Obviously, including debugging information into your executable will make your program larger, so when it comes time to deploy, submit, or send your program out for general use, you might want to recompile your code without debugging information.

I suggest you take a moment now and reflect on what happens when you "build" or "compile" a program into an executable and what it may mean to include debugging information into your running program.

Question: First, what are the steps in building a program into an executable? Second, describe those steps briefly.

Compiling with Debugging in Mind

To compile your code with debugging information, you will need to include the

```
      -g
```

compiler flag to the GNU gcc or g++ compilers. For instance,

```
      g++ -g -o myProgram file1.cpp
```

would cause the executable to be named *myProgram* (that's what the -o option does), and to have debugging information included with it (the -g option). While we're on the subject of compiling, I strongly suggest that you use the *-Wall* compiler flag. This option signals to the compiler to produce warnings as well as errors. I urge your to pay attention to the warnings just as you are forced to pay attention to errors. Programs submitted to me with warning will be docked points. If you really want to be forced to deal with warnings, you can use the *-Werror* option which will treat warnings just like errors.

Try it! Create a small Hello, World C++ program on our systems

sdand compile with and without debugging. Try playing around with the -Wall or -Werror flags.

Fix the Limits

By default, your accounts have been limited (you may or may not know that depending on your familiarity with UNIX systems). However, the limits I'm talking about relate to size of files you can have, the memory you can use, and more importantly for this topic, the core dump size you can handle.

Question: What is a core dump?

To view your limits, login to any of the Ubuntu machines accessible to you (your lab machines would work well) and type the following command:

```
ulimit -a
```

You will see something like the following, which is from my account and thus may be different from your account's limit listing:

```
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 126190
max locked memory (kbytes, -l) 64
max memory size (kbytes, -m) unlimited
open files (-n) 1024
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 8192
cpu time (seconds, -t) unlimited
max user processes (-u) 126190
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
```

With your limit information, pay attention to your core file size, for now. I believe it should state 0 just like the example above. You want to change this to be "unlimited". To do this, type

```
ulimit -c unlimited
```

The '-c' option is given to you in the original listing to show you how to make changes to the limits. For instance, to modify the max memory size you can use, you would utilize the -m command line argument. Back to the core file size. After making the change to make it unlimited, query all of your limits again to verify that the core file size is indeed unlimited. To make this permanent, you will need to add that line to your .profile, or other similar dot-file that is executed upon login.

Now, you may be asking yourself, why do I want to do this and what is a "*core dump*" file. As CS developers, you definitely want to do this, especially during the process of developing code. With the limit set to 0 for the core file size, you would have never had a "core" file show up in your account. Core files are dumps of the memory of the executable in the event of program failure (to be rather simple). For instance, if you are running your program (with debugging information compiled in) and you experience a segmentation fault, the operating system will generate a core dump and write it to a file. That file will be called *core*. You could then immediately inspect the core file with gdb and find out where and why your program segmentation faulted. Make sense?

NOTE: The ulimit command may not be available if you use the csh (C Shell). If you are using the csh shell (on Ubuntu, for instance), you will need to use the limit command. It is very similar in what it does, but the syntax is slightly different. You can view your limits on limit. To set your coredump file size on csh, you can type limit coredumpsize unlimited.

Question: So, now, what does the word *core* mean to you?

Try it! Now, write a *bad* program that accesses memory it shouldn't access. Compile it with the -g option, set your coredumpsize to unlimited, and run the program. You should get the following message:

```
Segmentation fault (core dumped)
```

and if you look in the directory where you ran your test program, you will see a file called *core*. If building a bad program is difficult for you, ask around the lab and see what others consider to be a bad program. If you want to find out more information about the core file and the options for naming it, take a look at the core man page (man core).

Be sure to turn in the code you wrote so I can see how you access memory poorly.

Running your executable with gdb

With your program ready to be debugged, this section will explain how you go about using gdb. gdb is a program that takes the name of the program to "debug" as it's primary command line argument. In other words, if your program was called myProgram, you would run it through gdb thusly,

```
gdb myProgram
```

which should result with the following (or something similar):

```
GNU gdb 6.8-debian Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html> This is free software: you are free
to change and redistribute it. There is NO WARRANTY, to the extent
permitted by law. Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu"... (gdb)
```

Your program is now ready to execute within the confines of the debugger. At this point, if you wanted to run your program, you would use the ***r*** command (short for "run") at the *(gdb)* prompt to start your program. If your executable takes command line arguments, you can enter them just as you normally would after the ***r*** command. For instance, with a couple of simple arguments:

```
(gdb) r --myarg 1 --myarg2 -v
```

would run your executable with the arguments that follow the ***r*** command.

What about that core file mentioned earlier?

Well, if your program generated a core file, you can use the core file with gdb to immediately find the location of the problem (in most cases). For instance, if myProgram generated a core file (called core in this example), you can tell gdb about the core when you run gdb, as follows:

```
gdb myProgram core
```

which will force gdb to inspect the core and give you information about the core dump.

Question: What happens when you run gdb with your program that has a bad memory access?

Using gdb

Once you have your program running with gdb, there are few commands that I will briefly go over that will help you with debugging.

- ***bt*** - Short for backtrace: this command will print a backtrace of all stack frames. This is the quickest method that I generally use to find out where a problem might have generated. It will show you the sequence of function calls that were called. I'd try it out.
- ***list*** - List a specified function or line: this command lets you see the code associated with the problem. list followed by a line number will list the file starting at that line number. list typed by itself will continue to print the following lines until the end of the file. list followed by the source file name, colon, and line number will print that file's information. For instance, line myProgram.cpp:84 will print line 84 (and following lines) from the myProgram.cpp file.
- ***print*** - print values from your variables: this command is quite useful and lets you inspect your variables values. The use of this command is quite intuitive. Try it out. You can dereference pointers, print structures, etc...
- ***break*** - short for breakpoint: this command is used for setting breakpoints in your code so that you can stop the execution of your program at appropriate times, inspect variables, and then continue on to other breakpoints, possibly stepping through the code as well. To set a breakpoint in your code, you type the break command followed by the line number. For instance, ***break*** *18* sets a breakpoint at line 18 of the current source. If you want to set a break point in a specific source file, you would type something like ***break*** *sourceFile.cpp:28*, to set a breakpoint at line 28 in sourceFile.cpp. Once breakpoints are set, you can re-run (or run) your program by using the ***r*** command (as described above). Once you run your program with breakpoints set, your code will stop executing at those breakpoints. You can then use the print or list commands to do inspection of your variables. The ***c*** command will help you get to the next breakpoint (or the end of the program).
- ***c*** - short for continue: continues running your program after a breakpoint.
- ***quit*** - pretty obvious; get out of gdb.

gdb is a highly useful tool. I suggest that you spend some time getting used to its most basic commands. Debugging your code in the manner shown above can be much more efficient than throwing all sorts of cout or printf statements all through the source.

Try It!

Use GDB with a simple program of your own. Write a different program other than the bad memory access program (or expand it a little). Try gdb out and get used to it. Set up a break point in your code, run your program, and inspect some variables.

Question: The question you should try to answer is to explain the difference between the ***n*** (short for next), ***s*** (short for step), and ***c*** (short for continue) commands with gdb.

Question: Using your bad program that accesses memory that it shouldn't or cannot access, compile your program with debugging information and run it. A core file should be generated (if you followed the above instructions). How large is the core file, as compared with the executable?

Question: How much larger are your sample programs with debugging information included as compared to not including debugging information?

Write up your results, and submit them to the moodle under this first assignment.

CMake - Cross platform build system

I believe it is important to write code that is cross-platform. That is, your code can execute on Windows, Linux, OS X, and other operating systems/architectures. Obviously, with some languages this is potentially easier (Java, for instance), but in reality it isn't that difficult with languages like C or C++ either. One of the big reasons I like this aside from the multi-platform execution is that you get to push your code through different compilers, operating systems, and hardware architectures. What this means for you and your code is that it makes it much more robust and correct. Compilers are different. Operating systems and hardware architectures are different. For instance, your code may have an error in it, but the Linux compiler set didn't generate any warnings or errors and the program appears to run fine on Linux. However, when compiled and run on OS X, it generates a runtime error. After inspection, you might find that you did have a bug in the code. This has been my experience with cross-platform coding.

Perhaps the biggest pain with cross-platform programming is navigating the different build systems. On UNIX/Linux you've got GNU compilers, Makefiles, and various IDEs (integrated development environments), like Eclipse. OS X has GNU compilers, Makefiles, and the XCode IDE. Meanwhile, Windows has the Visual Studio suite. Creating build systems for all operating systems/compiler pairs can take time and be a major pain, especially if you are not extremely familiar with a particular build system.

This is where CMake can help. CMake is a high level build system that can auto-generate specific build projects for different hardware/OS/compiler combinations. My objective in introducing CMake to you is not to provide a detailed understanding for how to use it, but rather give you a simple example (from the CMake documentation) and show you how you could use this build system. For more information about cmake, I will refer you to the following resources:

- http://www.cmake.org/ - Main CMake site
- http://www.cmake.org/cmake/resources/software.html - Downloads for CMake
- http://www.cmake.org/cmake/help/documentation.html - Specifics for the documentation of CMake

Simple Example

This example is based on a simple example from the CMake documentation, but it does a good job showing simple the simplicity of the configuration files. With CMake, you need to create a CMakeLists.txt file that describes your executable and its related source files. So, for a simple program with two source files: myProgram.cpp and helloDemo.cpp, you can describe your project using the following CMakeLists.txt file:

```
# The name of our project is "HELLO". CMakeLists files in this project
can

# refer to the root source directory of the project as
${HELLO_SOURCE_DIR} and

# to the root binary directory of the project as ${HELLO_BINARY_DIR}.



cmake_minimum_required (VERSION 2.8) project (HELLO)



# You can change the build type below. To have the Release
# build created, change Debug to Release set(CMAKE_BUILD_TYPE Debug)



# Add executable called "helloDemo" that is built from the source files

# "myClass.cpp" and "helloDemo.cpp".

add_executable (helloDemo myClass.cpp myClass.h helloDemo.cpp)
```

That's it, at least for a simple program. For more information and a slightly more interesting example, I'll refer you to the CMake Simple Example.

Now, how do you use your CMakeLists.txt file to create Makefiles, XCode project files, Eclipse projects, or VisualStudio solutions?

The Generators

Once you have a CMakeLists.txt file, you need to run the cmake program to have it generate a build system for you. For instance, on our Linux systems, you would probably want a Makefile to be generated. I tend to like my "builds" separated from the source so that things are easy to cleanup. To do this, I create a directory to hold the "build", and then, execute cmake from that directory to create the Makefile (or other build environment I'm working with). For example,

```
mkdir build
cd build
cmake ..
```

which would result in cmake reading the CMakeLists.txt file in the directory above it (where your source and the CMakeLists.txt file sit) to configure and generate the Makefile. You could then type

```
make
```

in the build directory and your program would build using the source files located in the directory structure above it. This works out rather nice as you can then later delete the build directory without affecting the source. Note that you only need to re-execute the cmake program if you change the CMakeLists.txt file. Once it generates the Makefile (or other build system) you can go about using the Makefile (or whatever IDE) as you normally would.

Alternatively, you can also invoke cmake with the "-i" option to get access to the various options that it sets. Using the example above, this would resemble the following:

```
mkdir build
cd build
cmake .. -i
```

Now, what about Xcode, Eclipse, KDevelop, or Visual Studio? To see which generators you can create aside from the default Makefile (on UNIX systems), you can run cmake as follows:

```
cmake --help
```

At the end of the listing you'll see the generators that can be used. To specify a generator, run cmake with the -G<GeneratorName> option. For instance, to generate the build system for KDevelop3, you would type

```
cmake . -G KDevelop3
```

This would generate a HELLO.kdevelop project file for your program. Similarly, for Xcode, you would type

```
cmake . -G Xcode
```

which would generate the related HELLO.xcodeproj Xcode project file for your program. One of the other nice things about creating a build directory is that you can multiple of them: one for XCode, one for VS 2010, and one for your UNIX Makefile build. To cleanup, you need only delete those directories. CMake does create other files for its use that can be deleted to regenerate. These are CMakeCache.txt, CMakeFiles, and cmake_install.cmake. However, these will also be contained in your build directories.

For Visual Studio, you will need to download CMake for Windows and do the generation on a Windows platform using the CMake gui.

With your build system generated, you can bring up the project with XCode (on OS X), bring it into KDevelop, or use the Makefiles on Linux to build the project.

CMake has many additional options for detecting libraries, and performing all sorts of checks that can help to make your code more portable. For that detail, I will refer you to the CMake documention.

Building a Library

CMake is also capable of handling multiple executables and multiple libraries all within the same CMakeLists.txt file. The example above only demonstrated a single executable. Here's an example that creates a library, called libmyLib.a that links to a couple of different executables.

```
cmake_minimum_required (VERSION 2.8)

project (HELLO)

# You can change the build type below. To have the Release

# build created, change Debug to Release set(CMAKE_BUILD_TYPE Debug)

# add library called libmyLib.a

add_library(myLib class1.cpp class1.h class2.cpp class2.h ... classN.cpp
classN.h )



# Add executable that links with the libmyLib.a library

add_executable (test1 test1_main.cpp)

target_link_libraries(test1 myLib)



# Add exectuable that links with the math library (libm.a)

# and the libmyLib.a library.
```

```
    add_exectuable (test2 test2_main.cpp)

    target_link_libraries(test2 myLib)

    target_link_libraries(test2 m)
```

In the above example, a library is created and then linked with multiple executables.

Try It!

Try it out with the simple example CMakeLists.txt file I created above. You'll need to supply some dummy code to make it work, but try it out. See if you can generate the Makefile, build, and then run your test program. If you feel like it and have access to another machine/architecture/OS, feel free to try another generator.

For this very simple exercise, create a single class file that has both a .h and .cpp component. Your class doesn't have to do to much of anything, but can be simple. For instance, initialize a variable to a value in the constructor and supply a get method to retrieve it. This class is the first (and only) component of your library. Next, create a main.cpp file that includes your class' .h file, creates an instance of the class, and prints the value you set. This is your executable.

Create a directory to contain these files, and create a CMakeLists.txt file (based on what I have here) that compiles and links your code. You should be able to run it.   Submit a tar'd (or zipped) file of your directory to the moodle for us to look over and test.