A GNU/Unix development environment primer

g++, make, and gdb

This handout presents the most useful aspects of using `gcc`/`g++`, `make`, and `gdb` in developing software. These utilities are not only extremely useful but also packed with features; this introduction almost completely exhausts the features that the author finds useful. These three programs constitute the most important elements of the GNU/Unix development environment.

The GNU project strives to produce a complete, freely available operating system. Though GNU stands for GNU's Not Unix, its development environment is in most respects identical to that of Unix, reflecting the Unix background of many of its programmers. In fact, nearly all of what this introduction covers about `g++` (the GNU C++ compiler) will apply also to `gcc` (the GNU C compiler) and `cc` (the Unix C compiler). Likewise, the behavior of the Unix debugger `dbx` is very similar to, though much less sophisticated than, the GNU debugger `gdb`. (Since Andrew machines do not use the GNU `make`, we will cover the Unix version.)

In future computer science courses you will likely find yourself using the GNU/Unix development environment. It is recommended, therefore, (but not required) that you become familiar with it now so that in later, bigger projects you are not hindered in learning a new development environment.

With practice, you will find the development environment convenient—even enjoyable—and you will come to understand why Unix is considered such a good operating system for software developers.

Your friendly command-line compiler, g++

The GNU compiler, `g++`, is all you ever really need to develop your C++ program. This section describes it and many of its options.

To compile a program, type `g++` followed by options and filenames. For instance, to compile a program named "prog.cpp", type "`g++ prog.cpp`" at the shell prompt. This will place in the same directory a file named "a.out", which will be the executable version of that program.

That is all there is to using `g++`. Its other capabilities come through command-line options. All of the following described options are important to learn. (There are many other, arguably less useful options.) You can, and will, use a combination of several options at once during development.

`-c` (*c*ompile only) The 'c' option tells `g++` to compile to object code only. The output will be the same filename with the suffix replaced by ".o", so the object code corresponding to "prog.cpp" will be "prog.o". (This can be changed with the 'o' option.) Object code can then be used to form the actual executable. For instance, if your program has three source files "foo.cpp", "bar.cpp", and "baz.cpp", you can compile each into its object code (typing, for instance, "`g++ -c foo.cpp`") and then compiling the object code together on a final invocation of `g++`: "`g++ foo.o bar.o baz.o`".

Object code can save compilation time, which can be prohibitive for large projects. By compiling source files into object code before linking the source files together to form the actual executable, you can avoid recompiling files that remain unchanged between compilations.

Using the 'c' option in conjunction with `make` (described below) makes for a powerful combination.

`-g` (debu*g*) Inserting the 'g' flag tells the compiler to insert more information about the source code into the executable than it normally would. This makes use of a debugger such as `gdb` much easier, since it will be able to refer to variable names that occur in the source code. This flag should always be included during development.

`-l` (load *l*ibrary) Loads a library during the object code link phase. Many precompiled libraries exist for your use. Most significant among these is the math library "libm.a", which contains the functions defined in <math.h>. (Other significant libraries include the curses terminal manipulation library "libcurses.a" and the X graphics library "libX11.a".) To compile a program which uses functions from the math library, type "`g++ foo.o -lm`". Note that the 'l' option follows the names of your files, and that you should omit the initial "lib" and the suffix ".a" from the library name (so that in the case of "libm.a", you get only "m").

`-o` (*o*utput) The 'o' flag denotes the destination filename of the output of `g++`. If you want your executable file to be named "prog", you might type "`g++ -o prog foo.o bar.o`".

`-D` (*D*efine) The 'D' flag allows you to define preprocessor constants on the command line. This is useful in debugging: you can set the `DEBUG` macro during the compilation of "foo.cpp" by typing "`g++ -DDEBUG foo.cpp`". To direct the preprocessor to replace occurrences of `DEBUG` with the value two, you would type "`g++ -DDEBUG=2 foo.cpp`".

`-O` (*O*ptimize) The 'O' flag tells `g++` that it should perform some optimizations on your code. Doing this can render runtime significantly faster, but it can also render compilation significantly slower, as optimization is a complex process. Generally—but not always—you should wait until your final debugging stages before setting this flag.

`-Wall` (*W*arnings *all*) The 'Wall' flag tells `g++` to turn on all warnings. These warnings frequently indicate possible coding errors that would be difficult to pin down as runtime errors. For this reason, you should *always* use the 'Wall' flag: "`g++ -Wall foo.cpp`".

The other part of the art of using `g++` is decoding its warnings and error messages, which at times can be downright cryptic. But that is subject matter for another handout.

Basic make

The `make` utility allows one to describe the compilation process concisely in a file named "Makefile" located in the same directory as your source code, so that to compile a program named "target" all you would type is "`make target`" at the shell prompt. At that point `make` would bring "target" up to date using the description of how in "Makefile".

An all-too-common approach to using `make` is to find somebody else's Makefile and to modify it appropriately to fit with your current project. Unfortunately, this strategy does not allow you to make much use of the utility's capabilities and scales up to larger projects very poorly. This short primer on the utility should describe enough to get you well on the way to writing your own sophisticated Makefiles.

Many other users write shell or Perl scripts to automate the compilation process, citing the non-portability of Makefiles. Though their argument is justified, their solution tends to be more complicated than necessary. Hopefully, the existence of a GNU `make` will allow the writing of more portable Makefiles. This introduction, though, covers Unix `make`, since that is what is available locally.

Explicit rules

The most basic of the Makefile rules is the *explicit rule*, which looks like the following:

```
prog: foo.cpp bar.cpp prog.h
	g++ -Wall -g -o prog foo.cpp bar.cpp
```

This rule has three parts. First we have the *target*, "prog", followed by a list of its *dependencies*, "foo.cpp", "bar.cpp", and "prog.h". These two parts are separated by a colon and are followed by a newline. The next part is a list of shell commands, each indented by a tab.

A target is called *out-of-date* if one of its dependencies is itself out of date or is newer than (i.e., has been modified more recently than) the target. The process of determining whether a target is out of date, therefore, is complex and recursive. In this example, "prog" will be out of date if any of "foo.cpp", "bar.cpp", or "prog.h" is newer than "prog". Note that source files such as "foo.cpp" have no dependencies, so in this example, since all the dependencies of "prog" are source files, `make` does not need to determine which dependencies are out of date.

If your Makefile includes the example rule above and you type "`make prog`", `make` will see if "prog" is out of date; if it is, it will update any out-of-date dependencies and then execute the shell commands following the rule.

Though this may sound quite complicated, in practice it is easy to understand: if "foo.cpp", "bar.cpp", or "prog.h" have been modified since "prog" was last compiled, the statement

```
        g++ -Wall -g -o prog foo.cpp bar.cpp
```

will be executed, making "prog" newer than all its dependencies. If it was already newer than its dependencies, there is no reason to recompile the program, so `make` will print a message explaining that there is nothing to compile and terminate.

Let us now look at a more complicated example Makefile (though its end result is the same thing as before):

```
prog: foo.o bar.o
        g++ -Wall -g -o prog foo.o bar.o

foo.o: foo.cpp prog.h
        g++ -Wall -g -c foo.cpp

bar.o: bar.cpp prog.h
        g++ -Wall -g -c bar.cpp
```

If you type "`make prog`" now, `make` will first update the dependencies of "prog" ("foo.o" and "bar.o") as necessary. Say that you have only modified "foo.cpp" since last making "prog". Then "bar.o" is still up to date, and only "foo.o" will be recompiled. Once "foo.o" is brought up to date, then "prog" will be brought up to date. So the process will execute the following two commands, if all goes well:

```
        g++ -Wall -g -c foo.cpp
        g++ -Wall -g -o prog foo.o bar.o
```

If there is an error in "foo.cpp", the process will terminate after the attempted compilation to "foo.o". If now you modify "prog.h" and then type "`make prog`", both "foo.o" and "bar.o" will be out of date, so the following command sequence will be executed.

```
        g++ -Wall -g -c foo.cpp
        g++ -Wall -g -c bar.cpp
        g++ -Wall -g -o prog foo.o bar.o
```

Note that compile-time errors (but not warnings) will terminate the make process, so if there is a problem with "bar.cpp", the final compilation in the above sequence will not occur.

This last example should give you an idea of how useful `make` can be in working with code spread over several files. We can now see, though, how the process can be made even simpler.

Quick hints

You can save command-line keystrokes by taking advantage of the fact that if `make` receives no arguments, it will use the first explicit rule in the Makefile. In the previous examples, then, if you type just "`make`", `make` will update "prog".

Also, frequently you want `make` to regard a source file (say, "foo.cpp") as changed when in fact you have not done anything to it. (This might be desirable if you have changed the Makefile rule or you have changed a header file that is not listed as a dependency.) You can quickly do this using the Unix `touch` command, which changes the modification time on a file without actually modifying the file: "`touch foo.cpp`".

Macros

Makefiles can contain *macros* which in behavior bear much resemblance to macros in C and C++; that is, their value is inserted using simple textual substitution. Defining a macro is easy: just separate the macro name from its value by an equals sign. For example, your Makefile might read:

```
CPP = g++
CPPFLAGS = -Wall -g -DDEBUG

prog: foo.cpp bar.cpp
        $(CPP) $(CPPFLAGS) -o prog foo.cpp bar.cpp
```

As you can see, after a macro `MACRO` is defined, you refer to it later as `$(MACRO)`. In this example, the statement

```
        g++ -Wall -g -DDEBUG -o prog foo.cpp bar.cpp
```

will be executed when you type "`make`" (providing either "foo.cpp" or "bar.cpp" has changed more recently than "prog").

This allows you to modify your Makefile's behavior more easily. If, after you finish debugging, you want to optimize the code for your final version, you need only change the definition of `CPPFLAGS` to:

```
CPPFLAGS = -Wall -O
```

To create your final version, type "`touch *.cpp`" at the shell prompt (so that all source code is newer than the object code and executables) and then type "`make`", which in this example results in the following compilation:

```
        g++ -Wall -O -o prog foo.cpp bar.cpp
```

Inference rules and suffixes

For large projects of many files, Makefiles can grow to unwieldy lengths. To deal with this, `make` allows for *inference rules*. Inference rules deal with transforming a file and changing the suffix, as happens during compilation into object code. An inference rule looks like this:

```
.c.o:
        $(CC) $(CFLAGS) -c $<
```

This rule says that if, as an example, "foo.o" does not have an explicit rule, then if "foo.c" exists, "foo.o" will depend on it and the command to create "foo.o" from "foo.c" will be (assuming that `$(CC)` is "`cc`" and `$(CFLAGS)` is "`-O`"):

```
        cc -c foo.c
```

This example uses a special macro, `$<`. This is valid only in inference rules and contains the out-of-date dependencies. (Other special macros are `$@`, which contains the name of the target, and `$*`, which contains the target's name minus its suffix (which is the same thing as the target's dependent's name minus its suffix).)

This seems magical, and indeed it nearly is. Using inference rules can bring down the length of a Makefile dramatically.

In fact, it is even more magical than it seems: `make` has many implicit inference rules, including the one we have already seen, to deal with code written in C, `yacc`, `lex`, and other tools. Unfortunately, `make` does not yet generally include an implicit inference inference rule to deal with C++ files. Let us then look at a Makefile for handling C++ files:

```
CPP = g++
CPPFLAGS = -Wall -g -DDEBUG

.SUFFIXES: .cpp

.cpp.o:
        $(CPP) $(CPPFLAGS) -c $<
```

This is much like the previous rule. The only difference is that, since `make` does not recognize ".cpp" as a suffix automatically, the ".SUFFIXES" line must be added to tell it about this new suffix.

A Makefile template

We mentioned earlier that many people get by with `make` by copying other Makefiles as templates and modifying it as necessary. In fact this strategy works for most simple cases. So here is a useful template for a Makefile:

```
# Define these macros to match with your project's filenames
TARGET = prog
OBJS = foo.o bar.o baz.o
LIBS = -lm
CPP = g++
CPPFLAGS = -Wall -g -DDEBUG

.SUFFIXES: .cpp

$(TARGET): $(OBJS)
        $(CPP) $(CPPFLAGS) -o $@ $(OBJS) $(LIBS)

# 'make clean' removes extra files that crop up during development
clean:
        rm $(OBJS) core

.cpp.o:
        $(CPP) $(CPPFLAGS) -c $<
```

(A comment in a Makefile begins with a hash sign ('#') and continue to the end of the line.) For most programs you write, you will find that you need only alter the macro definitions in the first few lines. You should know enough by now, though, to puzzle out how this works.

Debugging with gdb

So now you have your program compiled. How do you debug it? The simplest way is to edit the code frequently, telling it to print its status here, to print something when it reaches that point, etc. Doing this involves lots of trial and error, though, and a painful process of editing and recompiling. Much of this can be avoided by using `gdb`—the GNU debugger. The GNU debugger is a powerful successor to the Unix debugger `dbx`. It has an excellent command-line interface—which is what this primer will discuss—but it becomes particularly powerful within GNU's `emacs`.[1]

A graphical X-based version of `gdb`, named `xgdb`, exists, but only sporadically. Checking to see if it is on your particular machine is worthwhile.

Basics

Say you are debugging "prog". Compile "prog" with the 'g' flag (see the section on `g++` for more information about this flag), and fire up `gdb` by typing "`gdb prog`" at the shell prompt. This will load in the program, and from there you will get a command-line prompt, from then you can issue any of a number of commands.

help is the command to access `gdb`'s extensive online help system. With no arguments, this will describe the help system.

file changes the executable file being examined. If you type "`file foo`", then the executable under examination will become "foo".

quit exits you from `gdb`.

run begins your program at the beginning. Type what you would type at the shell prompt, but replace the executable's name with "`run`". For example, if you normally run your program by typing "`a.out 6 < inputfile > outputfile`", you should start it in `gdb` with the command "`run 6 < inputfile > outputfile`".

kill terminates the program. This is useful since the executable cannot be modified until `gdb` is done with it, so you cannot `make` your program until the program is terminated.

make works just like a call to `make` from the shell prompt. It saves you from going between the shell prompt and the `gdb` prompt all the time just to recompile your program.

list lists the ten lines surrounding the current line of the program. If you type "`list 100`", it lists the ten lines surrounding line 100.

Examining your program's state

When your program terminates abnormally, either due to an internal error or due to a user break from a control-C, or when your program is stopped due to reaching a breakpoint (described below) the program's state is maintained for examination within `gdb`. At this point you can examine it to help you determine where the termination occurred and what the variables' values were to make it stop there.

where displays the call stack.

[1] Thanks to Doug Rohde, whose Web document "An introduction to using gdb under Emacs" we used and even often pirated (with his permission) in preparing this section. (URL `http://www.cs.cmu.edu/~dr/gdbintro.html`)

up changes the function call under consideration to be the one above the current function call in the call stack. This allows you to investigate variable values in this newly current function call.

down moves the function call under consideration to be the one just below the current function in the call stack.

print displays the value of an expression in a readable format. The expression can be a single variable, such as `i`, or a more complicated expression, such as `*(A[i]->next)`. The expression could even tweak values: "`print i = 6`" will change the value of `i` to six. Or the expression can include a function call: "`print Fib(2)`". Be careful with assigning or calling a function call using `print`! The potential for messing something up is high.

Setting breakpoints

Normally `gdb` will run your program until abnormal or normal termination. You can investigate your program's behavior as it runs using breakpoints. The interface is not convenient, but it can work well if you have the patience.

break sets a breakpoint. Given a function name, `break` will set the breakpoint at calls to the function. If given a line number, it sets a breakpoint to the point when that line of source code is reached.

delete deletes the breakpoint whose number is specified by the argument. (A number is assigned to each breakpoint when it is created.)

step executes the current line of code. If the line contains a function call, `step` enters that function.

next executes the current line of code, skipping over code within function calls.

finish continues execution until the current function call is complete.

continue resumes execution of the program.

When you find yourself wanting to know about `g++`, `make`, or `gdb`, you will find the on-line man pages extensive and exhaustive (and perhaps exhausting).

Learning how to use `g++`, `make`, and `gdb` right can render the development process quicker, more effective, and more bearable. Of course nothing exists yet to make the development process trivial—but this is what makes programming fun. If you know how to wield it well, the GNU/Unix development environment will contribute to your programming experiences.