Make

The programs you will write as exercises in this specialization will be relatively small—at most a hundred lines and a couple of files. For programs of this size, recompiling the entire program takes a few seconds. Real programs are tend to be quite a bit larger. As the program size increases, so does the compilation time. As an example, one program changed, compiled, and run frequently by one of the authors has about 40,000 lines of code, 90 files, and takes about 75 seconds to compile completely.

While 75 seconds may not sound long, it is practically an eternity in the typical development cycle. Programmers recompile their code dozens to hundreds of times per day, meaning that all of those minutes add up—waiting 75 seconds 50 times in a day adds up to a bit more than an hour, or an eighth of a standard work day. This large of an overhead in the development cycle would be prohibitively large (despite its ability to provide convenient excuses for slacking off: e.g., http://xkcd.com/303/).

Fortunately, most of the time, one does not need to recompile all of the source code, if the object (*.o*) files from previous compilations are kept. Instead, only the file (or files) that were changed need to be recompiled, then the program needs to be linked again. Compiling each source file to an object file is accomplished with the -c option, and the various object files can then be listed on the command line to gcc in order to pass them to the linker. For comparison, recompiling one source file then relinking the same program described above takes about 1 second—much faster than recompiling the whole thing.

However, orchestrating this process manually is tedious and error-prone. If the programmer forgets to recompile a file, then the program will not work correctly, possibly in strange ways. Doing this manually not only requires the programmer to remember all files that were changed, but also which files *#include* files that were changed. For example, if the programmer changes myHeader.h, then any file which *#includes myHeader.h* must also be recompiled, as the source files' contents have *effectively* changed.

Long ago, programmers developed the make utility to not only automate this process, but also to simplify compiling programs in general. The make command reads a file called Makefile (though you can ask it to read an input file by a different name) which specifies how to compile your program. Specifically, it names the *targets* which can be made, their *dependencies* , and the *rules* to make the target.

When make is run, it starts from a particular target—something that it can build out of other things. A common starting target might be your entire program. make first checks if the target is up-to-date. Checking that a target is up-to-date requires checking that all of the files that the target depends on are themselves up-to-date, and that none of them are newer than the target. For example, the target for the whole program may depended on many object files. If any such file is not itself up-to-date (for example, the object file may depend on a . *c* file, which just got changed), it is rebuilt first. make then follows whatever rule was specified for the target to build it from its dependencies. If the target is already up to date, then it does nothing.

Using make to compile the program is also useful when the command line to compile the program becomes complex, especially when other people may need to do the compilation. Large complicated programs may require linking with several libraries, as well as a variety of other command line options. Trying to remember all of these and type them in every time you compile is painful. Even worse, many real programs need to be compiled by other people—whether it is multiple members of a large development team, or people who you distribute the program to. Expecting these other people to figure out the command line required to make your program leads to much frustration for them. Providing a *Makefile* allows anyone to build the program simply by typing *make* .