**Using the make program.**

Any C program of any size will normally be compiled not by invoking gcc from the command line but by using a program called make and a file called a "makefile". The name for a makefile is normally either Makefile or makefile (pretty imaginative, huh?). We will use Makefile for the name of the makefile; this makes it easier to see when you list the files in the directory.

The purpose of make is to simplify the process of rebuilding a program (*i.e.* a binary executable) from its source code, and to ensure that only source code that has been modified gets recompiled. This is not a big deal when you're dealing with a small number of source files, but when a program is split into tens or even hundreds of source code files (which is very common for large projects), it becomes a big deal very quickly.

make is a somewhat complicated program (it's really a miniature computer language of its own, completely distinct from C), so we will only cover the most rudimentary aspects of it here and refer you to the references when you need to know more. Much of the following material has been borrowed from the GNU make documentation. Also, you should realize that it isn't strictly necessary to use make when compiling C programs; it just makes the job much easier.

**Prerequisites**

You should understand the basics of compiling C programs. You should know the difference between source code files (.c and .h (header) files) and object files, and how to compile C programs in stages (first the object files, then the executable). If you are unclear on this material, read [this](http://www.cs.caltech.edu/courses/cs11/material/c/mike/misc/compiling_c.html).

**Writing a Makefile**

Using make is mostly about writing a Makefile, so we will discuss this first.

A Makefile consists mainly of:

* variable definitions
* rules, which are instructions for remaking files or performing some other actions. Rules themselves consist of:
  + targets
  + dependencies
  + commands

A **target** is usually the name of a file that is generated by a program; examples of targets are executables or object files. A target can also be the name of an action to carry out, such as clean, which normally is set up to remove unwanted files.

A **dependency** is another target that has to be dealt with before the current target is dealt with, and/or a file which the current target requires in order to be executed. Each target has a list of dependencies. Most of the time, dependencies are the names of other files that are used as input to create a particular target, either directly or indirectly. A target often depends on several files, and a single file may be a dependency for several other files.

A **command** is an action that the make program carries out when a specific rule is invoked. A rule may have more than one command, each on its own line directly below the rule. PLEASE NOTE: you need to put a tab character at the beginning of every command line! Forgetting to have the tab character at the beginning of each command line is the most common mistake in writing a Makefile. Note that four or eight spaces can not be used in place of the tab; it has to be the tab character itself (ascii 9 in hexadecimal). This is really lame, but it's the way make works.

Usually a command is in a rule with dependencies and serves to create a file with the same name as the target if any of the dependencies change. However, the rule that specifies commands for the target does not have to have dependencies. For example, the rule containing the commands associated with the target clean in the sample Makefile below does not have dependencies.

**A simple Makefile**

Here is a straightforward Makefile that describes the way an executable file called edit depends on eight object files which, in turn, depend on eight C source code files and three C header files.

In this example, all the C files include defs.h, but only those defining editing commands include command.h, and only low level files that change the editor buffer include buffer.h.

# Beginning of Makefile.

CC = gcc

edit: main.o kbd.o command.o display.o insert.o search.o files.o utils.o

$(CC) -o edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

main.o: main.c defs.h

$(CC) -c main.c

kbd.o : kbd.c defs.h command.h

$(CC) -c kbd.c

command.o: command.c defs.h command.h

$(CC) -c command.c

display.o: display.c defs.h buffer.h

$(CC) -c display.c

insert.o: insert.c defs.h buffer.h

$(CC) -c insert.c

search.o: search.c defs.h buffer.h

$(CC) -c search.c

files.o: files.c defs.h buffer.h command.h

$(CC) -c files.c

utils.o: utils.c defs.h

$(CC) -c utils.c

clean:

rm edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

# End of Makefile.

We split each long line into two lines using backslash-newline; this is like using one long line, but is easier to read. Note also that the # symbol means that the rest of the line is a comment. (This is different from comments in C; remember, a Makefile is not C code!)

As you can see, this Makefile contains ten different targets. One of them is the edit program, one is called clean, and the rest are C object code files (files ending in .o). There is also a single variable called CC (which by convention refers to the C compiler), which we set to be gcc. Variables are defined by writing

<variable-name> = <value>

and are used by writing $(<variable-name>) where needed.

Each rule has the form:

<target>: [<dependency1> <dependency2> ... ]

<tab>rule

<tab>rule

etc.

**Invoking make**

When you type:

% make

at the unix prompt (which is % here), the make program will look through the Makefile to find the first target in the file and then execute the commands appropriate for that target. This is known as the default target. If you want some other target, you have to specify it explicitly on the command line. For instance, to execute the commands for the "clean" target you would do

% make clean

In the Makefile shown above, the default target is edit, so typing make will cause the make program to try to rebuild the edit program.

When make starts rebuilding edit, the first thing it does is to determine whether it even has to rebuild it. To do this, it does the following:

1. For each of edit's dependencies, make checks to see if it's a file, and if so, if the file needs to be remade, and if so, remakes it (assuming there is a rule to remake it). If a file exists and there is no rule to remake the file (as is the case with source code files, for instance), make assumes that the file is up-to-date.
2. Assuming all of edit's dependencies are up-to-date, make checks to see if any of the dependency files have been modified more recently than edit itself has been. If so, it will execute the command(s) corresponding to the edit target. It does this by substituting variable values for variable references and then executing the resulting command(s).

If none of the dependency files have been modified more recently than the edit program itself, make will do nothing and will report that edit is up-to-date.

Note that if edit has never been compiled before, then make will try to compile it. If the name of the target is **not** the name of a file (*e.g.* the clean target), then make will always invoke the commands for that target when asked to make that target.

**What's the point?**

At this point you may be wondering why we need such a complicated system just to compile a few files. Here's why. Let's say that you modified the source code files command.c and command.h and want to recompile edit. What you don't want to do is to recompile every single source code file in the program. What you also don't want to do is to fail to recompile files that depend on either of these two files (for instance, kbd.o and files.o also depend on command.h). By letting make keep track of all the dependencies, you guarantee that when you modify some files, only the files that really need to be recompiled will be recompiled. This will usually only be a small fraction of the total number of source code files in a large project. For instance, if you modify one file in a project that has 1000 source code files (which is by no means rare), and ten other source code files in various directories depend on the file you modified, then only your file, the ten other source code files, and the final executable will be remade. That's obviously much faster than recompiling all 1000 source code files.

**Using variables to keep things concise**

Here is a shorter version of the Makefile above:

# Beginning of Makefile.

CC = gcc

OBJS = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit: $(OBJS)

$(CC) -o edit $(OBJS)

main.o: main.c defs.h

$(CC) -c main.c

kbd.o : kbd.c defs.h command.h

$(CC) -c kbd.c

command.o: command.c defs.h command.h

$(CC) -c command.c

display.o: display.c defs.h buffer.h

$(CC) -c display.c

insert.o: insert.c defs.h buffer.h

$(CC) -c insert.c

search.o: search.c defs.h buffer.h

$(CC) -c search.c

files.o: files.c defs.h buffer.h command.h

$(CC) -c files.c

utils.o: utils.c defs.h

$(CC) -c utils.c

clean:

rm edit $(OBJS)

# End of Makefile.

The only change is that we replaced the line main.o kbd.o command.o display.o insert.o search.o files.o utils.o (which occurred in three places) with $(OBJS) (which stands for "object files", although we could have used any name). This is convenient, because if we choose to change the name of one of the files, we only have to change it in the definition of OBJS and in the actual rule that makes the object file. The other uses of the file will read the variable definition and automatically get the new name. Defining variable names like this will make your Makefiles easier to manage.

**Finally...**

There is much, much more to make than we have time to go into here. Please consult the references or ask your helpful instructor if you want/need to know more.

**References**

* Andrew Oram, Managing Projects with Make. O'Reilly and Associates.
* The GNU Info documentation on make. Type "info make" at the unix prompt to access this.