## The make utility

As many of you probably already know, typing the entire command line to compile a program turns out to be a somewhat complicated and tedious affair. What the `make` utility does is to allow the programmer to write out a specification of all of the modules that go into creating an application, and how these modules need to be assembled to create the program. The `make` facility manages the execution of the necessary build commands (compiling, linking, loading etc.). In doing so, it also recognizes that only those files which have been changed need be rebuilt. Thus a properly constructed `makefile` can save a great deal of compilation time. Some people are "afraid" of `make` and its corresponding `makefile`, but in actuality creating a `makefile` is a pretty simple affair.

## Running make

Invoking the `make` program is really simple, just type `make` at the shell prompt, or if you are an EMACS aficionado `M-x compile` will do basically the same thing[ii]. Either of these commands will cause `make` to look in the current directory for a file called '`Makefile`' or '`makefile`' for the build instructions. If there is a problem building one of the targets along the way the error messages will appear on standard error or the EMACS '`compilation`' buffer if you invoked `make` from within EMACS.

## Makefile-craft

A `makefile` consists of a series of `make` variable definitions and dependency rules. A variable in a `makefile` is a name defined to represent some string of text. This works much like macro replacement in the C compiler's pre-processor. Variables are most often used to represent a list of directories to search, options for the compiler, and names of programs to run. A variable is "declared" when it is set to a value. For example, the line :

```
CC = g++
```

will create a variable named `CC`, and set its value to be `gcc`. The name of the variable is case sensitive, and traditionally `make` variable names are in all capital letters.
While it is possible to define your own variables there are some that are considered 'standard,' and using them along with the default rules makes writing a `makefile` much easier. For the purposes of this class the important variables are: CC, CFLAGS, and LDFLAGS.

| | |
|---|---|
| `CC` | The name of the C compiler, this will default to `cc` in most versions of `make`. Please make sure that you set this to be `gcc` or `g++` since `cc` is not ANSI compliant on the LaIR SparcStations. |
| `CFLAGS` | A list of options to pass on to the C compiler for all of your source files. This is commonly used to set the include path to include non-standard directories or build debugging versions, the `-I` and `-g` compiler flags. Sometimes this is called `CPPFLAGS` instead for C++ programs. |

[ii] 'M-x' means hold the 'meta' key down while hitting the 'x' key. If your keyboard does not have a 'meta' key then the 'ESC' will do the same thing. Hit the 'ESC' key and then the 'x' key. Do not hold down the 'ESC' key or else it will put you into eval mode.

| | |
|---|---|
| `LDFLAGS` | A list of options to pass on to the linker. This is most commonly used to set the library search path to non-standard directories and to include application specific library files, the -L and -l compiler flags. |

Referencing the value of a variable is done by having a `$` followed by the name of the variable within parenthesis or curly braces. For example:

```
CFLAGS = -g -I/usr/class/cs193d/include
$(CC) $(CFLAGS) -c binky.c
```

The first line sets the value of the variable `CFLAGS` to turn on debugging information and add the directory `/usr/class/cs193d/include` to the include file search path. The second line uses the value of the variable CC as the name of the compiler to use passing to it the compiler options set in the previous line. If you use a variable that has not been previously set in the makefile, `make` will use the empty definition, an empty string.
The second major component of makefiles are dependency/build rules. A rule tells how to make a target based on changes to a list of certain files. The ordering of the rules in the `makefile` does not make any difference, except that the first rule is considered to be the default rule. The default rule is the rule that will be invoked when `make` is called without arguments, the usual way. If, however, you know eaxctly which rule you want to invoke you can name it directly with an argument to `make`. For example, if my makefile had a rule for '`clean`,' the command line '`make clean`' would invoke the actions listed after the `clean` label, more on actions later.
A rule generally consists of two lines, a dependency list and a command list. Here is an example:

```
binky.o : binky.c binky.h akbar.h
<tab>$(CC) $(CFLAGS) -c binky.c
```

The first line says that the object file `binky.o` must be rebuilt whenever any of `binky.c`, `binky.h`, or `akbar.h` are changed. The target `binky.o` is said to depend on these three files. Basically, an object file depends on its source file and any non-system files that it includes.
The second line[iii] lists the commands that must be taken in order to rebuild `binky.o`, invoking the C compiler with whatever compiler options have been previously set. These lines must be indented with a `<tab>` character, just using spaces will not work! Because of this, make sure you are not in a mode which might substitute space characters for an actual tab. In particular, the "indented-text-mode" always "tabs" to the same indent level as the previous line, using spaces if the previous line were indented less than the standard full 8 spaces. This is also a problem when using copy/paste from some terminal programs. To check whether you have a tab character on that line, move to the beginning of that line and try to move "right" (`^f`). If the cursor skips 8 spaces to the right, you have a tab. If it moves space by space, then you need to delete the spaces and retype a tab character.
For “standard” compilations[iv], the second line can be omitted, and `make` will use the default build rule for the source file based on its extension, .c for C files. The default build rule that `make` uses for C files looks like this :

```
$(CC) $(CFLAGS) -c <source-file>
```

[iii] The second line can actually be more than one line if multiple commands need to be done for a single target. In this class, however, we will not be doing anything that requires multiple commands per target.

[iv] Most versions of make handle at least FORTRAN, C, and C++.

Here is a "complete" makefile for your reading pleasure.

```
CC = g++c
CFLAGS = -g -I/usr/class/cs193d/include
LDFLAGS = -L/usr/class/cs193d/lib -lgraph

PROG = example
HDRS = binky.h akbar.h defs.h
SRCS = main.c binky.cc akbar.cc
OBJS = main.o binky.o akbar.o

$(PROG) : $(OBJECTS)
 $(CC) -o $(PROG) $(LDFLAGS) $(OBJS)

clean :
 rm -f core $(PROG) $(OBJS)

TAGS : $(SRCS) $(HDRS)
 etags -t $(SRCS) $(HDRS)

main.o : binky.h akbar.h defs.h
binky.o : binky.h
akbar.o : akbar.h defs.h
```

This makefile includes two extra targets, in addition to building the executable: `clean` and `TAGS`. These are commonly included in makefiles to make your life as a programmer a little bit easier. The `clean` target is used to remove all of the object files and the executable so that you can start the build process from scratch[v], you will need to do this if you move to a system with a different architecture from where your object libraries were originally compiled, since source code is compiled differently on different types of machines. The `TAGS` rule creates a tag file that most Unix editors can use to search for symbol definitions[vi].

## Compiling in Emacs

The Emacs editor provides support for the compile process. To compile your code from Emacs, type M-x compile. You will be prompted for a compile command. If you have a makefile, just type make and hit return. The makefile will be read and the appropriate commands executed. The Emacs buffer will split at this point, and compile errors will be brought up in the newly created buffer. In order to go to the line where a compile error occurred, place the cursor on the line which contains the error message and hit ^c-^c. This will jump the cursor to the line in your code where the error occurred.

[v] It also removes any 'core' files that you might have lying around, not that there should be any.

[vi] Use 'M-x find-tag' or 'M-.' in emacs to search for a symbol within emacs. Tags are a useful thing.