Makefile Tutorial **taken from the web : http://min.ecn.purdue.edu/~rfisher/Tutorials/Make/**

This is an introduction and short tutorial on the use of **make**. It is intended to provide the student with enough understanding of **make** to write their own description files (makefiles) well enough to satisfy the needs of this course, and to facilitate the automation of the grading process for course projects.

Table of Contents

1. Introduction to **make**
2. Dependencies and the structure of makefiles
3. Dummy targets
4. Commands in a makefile rule
5. The use of variables in makefiles
6. Multi-level makefiles
7. Things you should know how to do

Bibliography

A good book on make is Andrew Oram and Steve Talbott's **Managing Projects with make**. It is published by O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95472. Copyright 1986, 1991.

This page was last modified September 04, 1997.

Overview of make:

make is a utility for organizing the files in a multi-file project. Its purpose is to help ensure that changes to a source file will be propagated to the parts of the project which depend on that file. **make** works by keeping track of which files a **target** depends on, so that if changes are made to any of these files, it will know that the target needs to be rebuilt.

To do this, **make** must have a listing of the files to be remade (the **target**s), the files upon which they depend (their **dependencies**), and the commands needed to build each target from its dependencies. This listing is called a **makefile**.

Before we get into the makefile itself, let's think about why we might have multiple files in our project...

Multiple source files, objects, and linking:

One common reason to use multiple source files is that writing code in multiple files allows functions to be split up into separate compiled **object** files which can be reused in other projects. This allows you to avoid rewriting similar code for each project, and to continually improve on the code you wrote earlier.

For example, you may decide to write one really useful set of math functions that you can use for the rest of your college career. Instead of rewriting the math code from scratch for each project or class, you can link in the object file containing the compiled versions of your math functions whenever you need one of these functions in a project.

Suppose you have written a file called **math.c** which contains the code for your math functions. You may want to compile this file to form an object called **math.o**, which could be included as part of a library or used on its own. This can be done with the following command:

```
cc -c -o math.o math.c
```

The "**-c**" tells **cc** to compile the source file **math.c** into an object (and more specifically, **not** to try to make it into an executable file). The "**-o math.o**" tells **cc** to call the object file **math.o**. Assuming there were no errors, this command will create an object file (for the machine that the compiler targets) that you can link with other objects to form executables use your math functions.

Suppose now that you want to write a program which uses some of these math functions, and that the main part of this program (the "main" function) resides in a file called **mainfile.c**. Within this file you call the function `add_em_up()` which is in your math object. You can compile **mainfile.c** into an object without having your math object available, but to make an executable (i.e. something that will run), you need to **link** the objects together. (Linking is the process of resolving the relative address of functions and variables across files.) This can be done in either one or two steps.

In the two-step version, the first step builds the mainfile object from **mainfile.c**, and the second links the mainfile and math objects to form the executable **myprog**:

```
cc -c -o mainfile.o mainfile.c
cc -o myprog mainfile.o math.o
```

To do this in one step you would use the command:

```
cc -o myprog mainfile.c math.o
```

This compiles **mainfile.c** into an object and passes that object along with the object **math.o** to the linker. The "**-o myprog**" tells **cc** to have the objects linked to form an executable called **myprog**. The linker resolves references to things like `add_em_up()` and forms the executable file. When this is done, **myprog** is a program just like **vi** or **ls** which can be run by typing **myprog** at the prompt and hitting <Enter>.

Dummy targets in makefiles

Not all targets used by "make" are files to be built. Some are merely named lists of other targets to be built. By skipping the command section of the rule, the dependencies will be updated, but no action will be taken afterward.

A common example is the target "all". This is often the first target in a makefile, and is usually just a list of dependencies which do the actual file building. For example, take the following makefile:

```
all: build install

build: myprog yourprog

myprog: mainfile.o math.o
        cc -o myprog mainfile.o math.o

mainfile.o: mainfile.c
        cc -c -o mainfile.o mainfile.c

math.o: math.c
        cc -c -o math.o math.c

yourprog: yourfile.c
        cc -o yourprog yourfile.c

install:
        cp myprog /usr/local/bin/myprog
```

In this makefile, there are two programs to be built, "**myprog**" and "**yourprog**". The targets "**mainfile.o**" and "**math.o**" just build objects which are to be linked to form "**myprog**". The target "**install**" is used to place the programs in the proper place in the directory tree.

The targets we are interested in are "**all**" and "**build**". Let's look at "**build**" first. The rule for "**build**" is:

```
        build: myprog yourprog
```

This rule causes "**make**" to update "**myprog**" and "**yourprog**". In other words, it will cause "**myprog**" and "**yourprog**" to be built. It is simply a list of the targets we want updated. By entering the command "`make build`" at the shell prompt, we cause "**make**" to build these two programs without us even knowing their names.

Notice that this rule doesn't have any commands, just a dependency list. Its entire purpose is to consolidate a set of targets to be built into one name.

Now, let's look at the rule for "**all**":

```
        all: build install
```

We see that this rule also has no command section. It's purpose is to provide a common target name to do everything that I (the user) might want the makefile to do.

Typically, when you build a program, you want to install it somewhere, so why not do it with one command? (Well, there are reasons, but sometimes they aren't too important.) In this example, typing "`make all`", will cause the programs to be built via the "**build**" rule, then installed via the "**install**" rule. With the "**all**" rule, I only have to type one command.

It turns out that I don't even need to type "`make all`". I could just type "`make`" and **make** will start with the target "**all**" just as if I had typed "`make all`". This is because "**make**" takes the first target listed in the

makefile as its "**default**" target if no target is given on the command line.

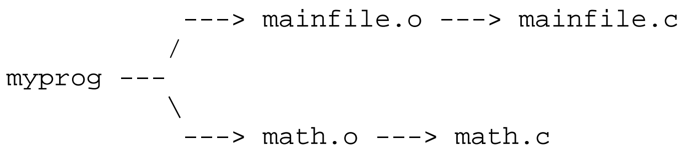
If we just type "`make`" at the command line, "**make**" will scan through the makefile and find the first target; in this case, "**all**". It will then start processing the rule for "**all**", by processing each of its dependencies, each of their dependencies, etc. So, using this makefile, the user can build and install the programs "**myprog**" and "**yourprog**" without knowing their names by simply entering "`make`" at the command line.

Dependencies:

In the example in lesson 1, we made a program called **myprog** from two source files: **mainfile.c** and **math.c**. **math.c** contains the code for a set of math functions which we reference in **mainfile.c**. Suppose we have now found a bug in our code for the function `add_em_up()`. If we change the code in **math.c**, we need to recompile the object **math.o** and the executable **myprog** in order for the bug to be fixed in our program. We do not have to recompile **mainfile.c** to make **mainfile.o** again, because **mainfile.o** only contains **references** to the functions in **math.o**, not the functions themselves. We only need to relink the two objects into an executable after fixing **math.o** in order to propagate the bug fix to **myprog**.

Recall (from lesson 1) that **myprog** must be built from **mainfile.o** and **math.o**, that **mainfile.o** is built from **mainfile.c**, and that **math.o** is built from **math.c**. We say that **myprog** "depends" on **mainfile.o** and **math.o**, that **mainfile.o** "depends" on **mainfile.c**, and that **math.o** "depends" on **math.c**.

We can show this set of relationships in graph form:



In this graph, the arrows (Okay, so it isn't artwork. Tough.) show the dependencies between the files. Changes must be propagated against the direction of the arrows. Thus, we see that a change in **mainfile.c** will cause **mainfile.o** to be "out-of-date", which will in turn cause **myprog** to be out-of-date. We can also see that this change will not affect **math.o** or **math.c**. Similarly, a change in **math.c** will cause **math.o** and **myprog** to be out-of-date, but not affect **mainfile.o** or **mainfile.c**.

This shows that splitting the program into two files not only allows us to easily reuse the math object in other projects, but it saves us from a complete recompile of the entire project if only one function is changed.

Makefile representation of a target and its dependencies:

Above, we showed that using multiple files in a project can be beneficial, but it can also be a pain in the neck. Keeping track of which object was built when, and wether or not a particular executable depends on something that was just changed can be a real hassle. It is for this reason that **make** was created. **make** reads a file called a "makefile" (usually named **makefile** or **Makefile**) which lists the project's executables and/or objects along with the files upon which they depend. It also contains lists of commands to execute in order to rebuild dependant files from their dependencies.

Let's look at the part of the makefile for **myprog** above:

```
myprog: mainfile.o math.o
        cc -o myprog mainfile.o math.o
```

For the moment, we won't worry about the rest of the makefile. The first part is the word **myprog** followed by a colon. This defines the "target". The target is the file we want built in some manner (for our purposes, by compiling other files). The rest of the rule describes the target's dependencies and the commands for building the target from these dependencies.

After the colon, we see **mainfile.o** and **math.o** These are the files on which **myprog** depends. When make is told to build **myprog** it will check to see if these files are up to date (more on that later). If they are, and **myprog** is newer than these files, make will assume that **myprog** doesn't need to be rebuilt. Otherwise, it will make sure that each of these files is up-to-date by finding the rules which define their dependencies, etc.

Once all the dependencies have been brought up to date, make will execute the command on the second line of the rule to update **myprog** The second line must begin with a tab character (no spaces!), and contains a command that could be typed at the shell prompt to make the target from the dependencies. In this rule, the command is "`cc -o myprog mainfile.o math.o`". This is the command we saw above

which links the object into an executable file.

Suppose **mainfile.o** doesn't exist when we tell make to build myprog, then the target for **mainfile.o** is found, its dependencies are checked and rebuilt if necessary, and the commands to build **mainfile.o** are executed. To build **mainfile.o** we need to compile **mailfile.c** into an object:

```
mainfile.o: mainfile.c
        cc -c -o mainfile.o mainfile.c
```

We see that this just encodes the dependency of **mainfile.o** on **mainfile.c**, and that you would use the command "`cc -c -o mainfile.o mainfile.c`" to build **mainfile.o** from **mainfile.c**.

For math.o, we would have the rule:

```
math.o: math.c
        cc -c -o math.o math.c
```

Putting these rules all together, we have the makefile:

```
myprog: mainfile.o math.o
        cc -o myprog mainfile.o math.o

mainfile.o: mainfile.c
        cc -c -o mainfile.o mainfile.c

math.o: math.c
        cc -c -o math.o math.c
```

Let's look at the graph we made earlier showing the dependencies of the files:

```
We can show this set of relationships in graph form:

           ---> mainfile.o ---> mainfile.c
          /
myprog ---
          \
           ---> math.o ---> math.c
```

If we type "`make myprog`" at the command prompt, **make** will walk the graph we showed above to find all the dependencies that have be changed, then walk back up the graph (against the arrows) updating targets until it rebuilds **myprog**

The command section of a rule

The commands within a rule are executed by "**make**" whenever "**make**" determines that the target must be rebuilt. The command section of a rule is a list of one or more commands that will be passed to the shell by "**make**".

So far, we have only looked at make rules with one command. For example:

```
project2: proj2.c support.c myheader.h
        gcc -o project2.c support.c
```

In this rule, the only command necessary to build "**project2**" is:

```
        gcc -o project2.c support.c"
```

However, we may need to execute more than one command to build the target from its dependencies. For example, the documentary part of my Preliminary Exam is written in LaTeX. (Hank hates this, he likes **nroff**.) To build the so-called "Device Independent" file (which is later converted to Postscript for printing), I run LaTeX three times and BibTex once (so that LaTeX can resolve references that it couldn't in previous passes).

The command section of a "**make**" rule is the set of lines immediately following the target line which start with a tab character. The first line which does not start with a tab signals "make" that there are no more commands for that rule. For my Prelim, the following rule suffices:

```
prelim.dvi:     prelim.tex ../Bib/prelim.bib
        latex prelim
        bibtex prelim
        latex prelim
        latex prelim
```

When "**make**" decides that "**prelim.dvi**" must be rebuilt, it will execute the commands "`latex prelim`", "`bibtex prelim`", "`latex prelim`", and "`latex prelim`", one at a time, in order from top to bottom.

If one of the commands fails, "**make**" will stop and report an error without processing the remaining commands or the rest of the rules necessay to build the ultimate target. "**make**" can be told to ignore an error from a command and continue with the remaining commands (and with the rest of the make process) by placing a '-' character before the first word of the command (just after the tab character):

```
prelim.dvi:     prelim.tex ../Bib/prelim.bib
        -latex prelim
        bibtex prelim
        latex prelim
        latex prelim
```

In this rule, "**make**" is told to continue even if the first execution of LaTeX fails. This example is not a case where this feature would normally be used though, because the later commands really do require the first command to work properly. You are much more likely to see something like this:

```
test:
        -mkdir ./test
        cp myproj test
        test/test < inputfile > test/outputfile
```

Here, "`mkdir ./test`" will fail if the subdirectory "**test**" already exists. Our actual goal in calling this command was to be sure that "**test**" exists, not to be sure that the command was successful. We don't care if "`mkdir`" failed, as long as "**test**" exists before we try to copy "**myproj**" to it. Placing the dash before "`mkdir ./test`" keeps us from having to remove the file "**test**" before executing the commands for this rule.

Variables in makefiles

Variables in a makefile work much the same as variables in a shell script. They are words to which a string of characters can be assigned. Once a string has been assigned to the variable, every reference to the variable in the rest of the makefile is replaced by the string. Variable names are usually chosen to be in all capital letters for clarity.

The variable is assigned a string value via an assignment statement, such as

```
        MYFILES=a b c d e
or
        OUTFILE=myfile.c
```

The first assignment above sets the variable "*MYFILES* " to the character string "**a b c d e**". The second assignment sets "*OUTFILE* " to the character string "**myfile.c**". Variables may also be left unset in the makefile. In this case they are implicitly set to the empty string "". For example,

```
        CFLAGS=
```

sets "*CFLAGS* " to the string "". Referencing a variable before making an assignment to it leaves it set to the empty string until the assignment is made. So references to a variable before its first assignment will be replaced with "". References after the assignment will be replaced with the newly assigned string.

Variables are referenced by placing a dollar sign ($) before the variable name and placing the name in braces or parenthesis depending on the version of "**make**" you are using. Most versions allow you to use just the dollar sign in certain cases, but some require a pair of delimiters (braces, parens, etc) around the name. Therefore, it is safest to always enclose the name in a delimiter pair. Braces and parenthesis also have different meanings in some versions of "make", so you may have to use a particular type of delimeter. See the man page for the version of "**make**" which you are using. I use GNU make, so I'll just use braces {} for the rest of this tutorial.

Suppose at the top of our makefile we have the line:

```
        MYFILES=a b c d e
```

To reference MYFILES later in the makefile we could write:

```
        ${MYFILES}
```

which would be translated to the string "a b c d e".

Variables may also be used within other variable assignments:

```
BASEDIR=/usr/local
HOMEDIR=${BASEDIR}/pccts
BINDIR=${BASEDIR}/bin
```

This code segment sets *BASEDIR* to "/usr/local", *HOMEDIR* to "/usr/local/pccts", and *BINDIR* to "/usr/local/bin".

Finally, here is a short example of a makefile using variables:

```
CC=/bin/gcc
```

```
CFLAGS=-Wall
INSTDIR=/usr/local/bin

${INSTDIR}program: myfile.c
        ${CC} ${CFLAGS} -o ${INSTDIR}/program myfile.c
```

In this example, after making the string substitutions, the command that "make" will use to build "program" from "myfile.c" is:

```
        /bin/gcc -Wall -o /usr/local/bin/program myfile.c
```

This allows the user to customize which compiler is used, the options to the compiler, and the directory the executable will be installed in just by changing the strings assigned to the variables.

Multilevel makefiles

The command section of a makefile rule may recursively call "make". We can use this to organize a multipart project, or set of projects, into a set of subdirectories which each have their own makefile.

Suppose, for example, that we have a project which has separate UDP and TCP sections. We would like to keep the code for each of these sections in different subdirectories like so:

```
                  ---udp
                 /
        project<
                 \
                  ---tcp
```

We will put a makefile in each of the directories: project, udp, and tcp. The job of the top-level makefile is to direct the operation of the lower-level makefiles. Their job is to build the udp and tcp versions of the project. The top makefile might look like this:

```
all:
        (cd udp; make all)
        (cd tcp; make all)

install:
        (cd udp; make install)
        (cd tcp; make install)

clean:
        (cd udp; make clean)
        (cd tcp; make clean)
```

The first command for the target "all", starts a subshell which goes into the "udp" subdirectory then runs "`make all`". That "make" will build the UDP version then return to the top-level makefile. The second command for "all" then starts a subshell which goes into the "tcp" subdirectory and runs "`make all`". This "make" will build the TCP version of the project before returning to the top-level makefile. How the UDP and TCP versions are actually built are hidden from the top-level makefile, and can be changed without changing the top-level makefile. (Except that they must provide the targets used by this makefile.)

The makefile for the UDP version might look like this:

```
INSTDIR=/usr/local/bin

all: server client

server: server.c
        gcc -Wall -o server server.c

client: client.c
        gcc -Wall -o client client.c

install: server client
        mv server client ${INSTDIR}

clean:
        -rm -f server.o client.o core
```

When the top-level makefile runs `(cd udp; make all)`, this makefile will build **server** and **client** in the udp subdirectory. Running `(cd udp; make install)`, from the top level makefile will cause this makefile to move the executables in /usr/local/bin. Also, running `(cd udp; make clean)` from the top-level makefile will cause the objects and a possible core file to be removed from the udp subdirectory.

Another reason for using multi-level makefiles is to isolate user-customizable variable settings to the top-level makefile. This lets the user work exclusively with the top-level makefile, rather than having to go into each subdirectoy to change variable values. The top-level makefile then passes variable values to the lower-level makefiles which need them.

The UDP makefile above appears to violate this idea because it sets INSTDIR explicitly, but a variable setting from the command line takes precedence over a setting in the makefile. By changing the "install" rule in the top-level makefile to pass a value for INSTDIR to one or both of the lower-level makefiles, we can isolate the customization to the top level:

```
all:
        (cd udp; make all)
        (cd tcp; make all)

install:
        (cd udp; make install INSTDIR=/usr/local/bin)
        (cd tcp; make install INSTDIR=/usr/local/bin)

clean:
        (cd udp; make clean)
        (cd tcp; make clean)
```

Now the user just hase to change the values passed as INSTDIR. We can tighten this a little by using a variable at the top level:

```
INSTDIR=/usr/bin

all:
        (cd udp; make all)
        (cd tcp; make all)

install:
        (cd udp; make install INSTDIR=${INSTDIR})
        (cd tcp; make install INSTDIR=${INSTDIR})

clean:
        (cd udp; make clean)
        (cd tcp; make clean)
```

Note that the commands in the "install" rule are interpreted as:

```
        (cd udp; make install INSTDIR=/usr/bin)
        (cd tcp; make install INSTDIR=/usr/bin)
```

These commands will run "make" in each of the subdirectories and use "/usr/bin" as INSTDIR in each, even if INSTDIR is set in the lower-level makefile. If we run:

```
        (cd udp; make install)
```

"make" will use INSTDIR as defined in "udp/makefile" because it is not overridden by the top-level makefile. Also, if we go into the udp subdirectory and run make from there (without using the top-level makefile), INSTDIR will be defined as in "udp/makefile". This is one reason that running lower-level makefiles by hand should be discouraged.

Forward

The purpose of this tutorial is to ensure your ability to write a makefile to facilitate the testing and grading of your projects. We will look at a few topics.

Renaming files to match project specifications

When a project is assigned to you, we usually will specify the name of the program you are to create. For example, if you are building a parser, we may tell you that it must be named "parser" and that the main source file must be "parser.c". Sometimes, we change our minds about the names of the files, or don't come up with one until after you have started. (Okay, sometimes we're just plain sloppy with our project specs.) Get used to it, this is life. Real customers and bosses will probably be even more idiotic in the changes they make, and usually aren't as good at specifying what they want in the first place. I'm not kidding, I've been there.

So, let's say that you have been building a parser from the file "myparse.c", and have been building an executable called "myparse". Your makefile may look something like this:

```
myparse: myparse.c support.o
        cc -o myparse myparse.c support.o

support.o: support.c
        cc -c -o support.o support.c
```

Now, the evil Randy comes along and says "I want your main project source file to be named `parser.c' and the executable to be named `fred'." Why? Because he is a petty, vindictive, and merciless TA, that's why. It doesn't matter if it makes no sense: the grader gets what he wants, or you get a 0.

You now have two choices:

1. Panic. In which case Randy will just sit back and watch the show.
2. Think about it, then change something.

I'm quite sure you have all had enough practice to handle the first option without any further instruction, so let's try the second. If we have written a decent makefile, we should be able to change the rules to use the new names.

The original makefile was:

```
myparse: myparse.c support.o
        cc -o myparse myparse.c support.o

support.o: support.c
        cc -c -o support.o support.c
```

The program needs to be called "fred" rather than "myparse", so we can simply change the rule to generate "myparse". In this rule we change the target so that "make" looks for "fred" rather than "myparse" when deciding if to build it. We also need to change the command to build "myparse" so that it builds "fred" instead. The new rule looks like this:

```
fred: myparse.c support.o
        cc -o fred myparse.c support.o
```

That was simple. Now we need to figure out how to handle Randy's idiotic decree about the name of our source file. Essentially, we must change "myparse.c" to "parser.c" everywhere it appears in the makefile. This isn't really that difficult either. In this example we only have two places where "myparse.c" is mentioned, both in the rule for what is now "fred":

```
fred: parser.c support.o
        cc -o fred parser.c support.o
```

Our new makefile looks like this:

```
fred: parser.c support.o
        cc -o fred parser.c support.o

support.o: support.c
        cc -c -o support.o support.c
```

Of course, you will have to rename the source file itself, which may cause other problems, but if you don't panic, and don't build overly complicated makefiles, you should be able to handle it.

For the record it is unlikely that we will have you change the name of a file on short notice. It has happened before though, so you should be able to handle the situation.

Let's look at another solution to the same problem. This time, we will add a rule to the makefile instead of changing the current rules. The original makefile was again:

```
myparse: myparse.c support.o
        cc -o myparse myparse.c support.o

support.o: support.c
        cc -c -o support.o support.c
```

We need to have the executable "fred" made, but there is no reason we can't make "myparse" instead and then rename it. So let's add a rule to do just that:

```
fred: myparse
        mv myparse fred

myparse: myparse.c support.o
        cc -o myparse myparse.c support.o

support.o: support.c
        cc -c -o support.o support.c
```

Now, when we build "fred", we actually build "myparse" then rename it to "fred". Easy enough. No panic necessary. We can do the same type of thing for "parser.c" and "myparse.c" by adding a rule that copies "parser.c" to "myparse.c":

```
fred: myparse
        mv myparse fred

myparse.c: parser.c
        cp parser.c myparse.c

myparse: myparse.c support.o
        cc -o myparse myparse.c support.o

support.o: support.c
        cc -c -o support.o support.c
```

Notice that we didn't **move** parser.c to myparse.c, we just have make copy it whenever we change parser.c. We still have to rename the source file from "myparse.c" to "parser.c" (because Randy said so, remember),

but we don't have to search the entire makefile for each occurence of "myparser.c".

The last solution was inefficient though (from a filespace perspective), so we might try to use another tool that we have learned about: variables.

As we write the file, we could use variables to refer to the programs we wish to build and their sources. Suppose we use EXECFILE and SRCFILE to represent these files respectively. We would then have the following original makefile:

```
EXECFILE=myparse
SRCFILE=myparse.c

${EXECFILE}: ${SRCFILE} support.o
        cc -o ${EXECFILE} ${SRCFILE} support.o

support.o: support.c
        cc -c -o support.o support.c
```

Changing the file to build "fred" from "parser.c" only requires a change in the definition of the variables EXECFILE and SRCFILE:

```
EXECFILE=fred
SRCFILE=parser.c

${EXECFILE}: ${SRCFILE} support.o
        cc -o ${EXECFILE} ${SRCFILE}support.o

support.o: support.c
        cc -c -o support.o support.c
```

This is easy when first writing the makefile, and makes changes easier later, but is more work than using the first solution once the makefile is written without using the variables.