

CSI605

Introduction to make

Advantages of Make

- **Significantly reduces the amount of time spent compiling a program.**
- **Insures that programs are compiled with correct options.**
- **Insures that programs are linked to the current version of program modules and libraries**
- **Frees the user of having to remember all of the grungy details each and every time**

Example of Why Use Make

- **Scenario**
 - **Program.c was just modified**
 - **Desire to compile and link to the modules inputs.c and outputs.c**
- **Approach 1**
 - **gcc program.c inputs.c and ouputs.c**
- **Approach 2**
 - **gcc program.c inputs.o outputs.o**
 - **Takes less time**
 - **Relies on programmer to assure timeliness of inputs.o and outputs.o**

What make does?

- **Determines whether the relevant object files exist and are current.**
- **Performs the smallest set of compilations necessary.**
- **Increased efficiency and less room for error.**

make Terminology

- **Target**
 - **A task that needs to be performed. Often the name of a file that you wish to create.**
- **Dependency**
 - **A relationship between two targets. If changes in target B produces changes in target A then A depends on B.**
 - **Example**
 - **vector.o depends on vector.c and vector.h**
- **Up to date**
 - **A file that is more recent than of the files on which it depends.**
 - **Example**
 - **vector.o is more recent than vector.c and vector.h then vector.o is up to date**
- **makefile**
 - **This is a file that describes how to creat one or more targets. It lists dependencies and rules for compiling the target.**
 - **One typically wants a sigle makefile in each directory named makefile or Makefile.**

Creating a makefile

Example makefile number 1

#CSI605 Fall 99

#Please note that targets begin in the left column followed

#by :

#Also note that shell command lines must begin with a tab.

myprogram:

Here we place the commands to create myprogram

gcc -o myprogram -O myprogram.c vector.c scaler.c

myprogram.db

Here we place the commands to create myprogram.db

gcc -DDEBUG -g -o myprgram.db myprogram.c vector.c
scaler.c

- This example makefile merely contains the commands to create two different versions of myprogram
- Invocation
 - make myprogram
 - make myprogram.db
 - make

Extending Commands

- Use `\` to extend the command across several lines
- **Scope of UNIX characters**
 - `cd ../myprogrsource`
 - `gcc myprogrsource.c`
 - `This is FOOBAR`
- `cd ../myprogrsource;gcc myprogrsource`

Dependencies

- Here is the first part of our make file redone with explicit and implicit dependencies

myprogram: myprogram.o vector.o scaler.o

gcc -o myprogram myprogram.o vector.o scaler.o

myprogram.o: myprogram.c

gcc -c -O myprogram.c

vector.o: vector.c headerfile.c

gcc -c -O vector.c

scaler.o: scaler.c

gcc -c -O scaler.c

Recursive Checking

- **Change scaler.c then what happens?**
 - **Determines whether myprogram.o, vector.o, and scaler.o are themselves targets.**
 - **Once it discovers that they are targets then it checks to see if they are up to date relative to their sources.**
 - **Determines that scaler.o is older than scaler.c and compiles a new version.**
 - **Checks myprogram's date relative to myprogram.o, vector.o, and the new version of scaler.o**
 - **Determines that myprogram is out of date with regard to scaler.o.**
 - **Creates a new version of myprogram by linking the object files.**
- **We note that a target if it does not exist as a file is out of date.**
- **Make task always executes the task**

make clean

- When we encounter make clean we want to delete everything
- Recall that the target names that are not files are always out of date

clean:

rm *.o *.do stimulate stimulate.db

- Another good use of this approach is the creation of a target called install that moves the final executable to an appropriate home and sets the access modes appropriately

Abbreviations

- We can use abbreviations to help make writing make files easier
 - `$$` stands for the full name of the target
 - `$$*` stands for the name of the target with the suffix deleted

- Standard way

stimulate: stimulate.o inputs.o outputs.o

gcc -o stimulate stimulate.o inputs.o outputs.o

inputs.o: inputs.c headerfile.c

gcc -c -O inputs.c

- With abbreviations

stimulate: stimulate.o inputs.o outputs.o

gcc -o \$\$ \$\$*.o inputs.o outputs.o

inputs.o: inputs.c headerfile.c

gcc -c -O \$\$*.c

Macros

- macro-name = macro-body
- make processes this by substituting macro-body for the string `$(macro-name)`
- Considering our previous example we write
DEPENDS = inputs.o outputs.o stimulate.o
stimulate: \$(DEPENDS)
gcc -o \$@ \$(DEPENDS)
- Macros may also be used to hold compilation options
CFLAGS = -DDEBUG -g
gc -c \$(CFLAGS) inputs.c

Search Directories

- **make normally looks for files in the current directory**
- **We may modify this behavior with the use of VPATH**

VPATH = dir1:dir2:...:dirn

- *Not all versions of make support a robust version of VPATH*
- **VPATH can be used to help one organize their work**

Default Compilation Rules

- **We can actually allow make to use default rules to automatically compile things for us**
- **Two rule sets**
 - **Suffix rules**
 - **Work on all versions of UNIX and MS-DOS**
 - **Pattern rules**
 - **More flexible and the method of choice under GNU**
- **We will examine each separately**

Suffix Rules

- **First step is to clear the suffixes**
.SUFFIXES
- **Next list suffixes involved in rules**
.SUFFIXES .c .o .do
- **Specify suffix rule**
 - **List the suffix of the file that make should look for**
 - **eg .c file**
 - **Followed by the suffix of the file that make should build**
 - **eg .o file**
 - **Follow with a colon then a semicolon**
 - **End with a UNIX command stated in general terms**
 - **.c.o:; gcc -c -o \$@ -O \$*.c**
 - **We may override this by including a standard entry that builds a particular .o file**
- **Alternate form**
.c.o:; gcc -c -o \$@ -O \$<
- **Debug form**
 - **.c.do:; gcc -c -o \$@ \$(CFLAGS) \$<**

Putting it All Together

```
#Clear Suffixes
.SUFFIXES
#Specify suffix rules for .c, .o, and .do
.SUFFIXES: .c .o .do
DEBUGFLAGS = -DDEBUG -g
PRODUCTFLAGS = -w -o
PRODUCTOBS = program.o other.o
DEBUGOBS = program.do other.do
EXECNAME = product
#Suffix rule to make a "production" .o file
.c.o:; gcc -c -o @$ $(PRODUCTFLAGS) $<
#A suffix rule to make a .do file (debugging .o module)
.c.do:; gcc -c -o @$ $(DEBUGFLAGS) $<
#Generate executables:
production: $(PRODUCTOBS)
    gcc -o $(EXECNAME) $(PRODUCTOBS)
debug: $(DEBUGOBS)
    gcc -o $(EXECNAME) $(DEBUGOBS)
#Target line to make some object modules
program.o program.c header.h
other.o other.c
#Target lines to make some debugging object modules.
Program.do program.c header.h
other.do other.c
clean:
    rm *.o *.do *~
    rm product
```

This makefile in action

make program.o

- Checks the date of program.c and uses the suffix rule for creating a .o file if necessary
- Resultant compilation

gcc -c program.o -w -O program.c

make other.o

- Resultant compilation

gcc -c other.o -w -O other.c

make other.do

- Creates a debugging version of this program

make production

- Makes production by linking and if necessary compiling the necessary files

make debug

- Makes debug by linking and if necessary compiling the necessary files

make clean

- Deletes all object modules and executables

Pattern Rules

- This look like normal makefile entries except they contain % signs as wildcards.

%o.o : %o.c

gcc \$(CFLAGS) -c \$<

- This means that any object file can be built by replacing the .o in the filename with the .c and compiling the file with the resulting name

- makes default rules

program.o: program.c

- Resultant compilation
 - cc -c program.c -o program.o
- To force the previous default rule to use gcc instead of cc merely set in the make file

CC = gcc

- This works because the default make rule uses the \$(CC) macro
- Extreme versions of default makefiles
 - make program.o
 - Resultant compilation in the absence of a makefile
cc -c program.c -o program.o

Invoking make

- **make target-name**
 - **make reads the makefile in the current directory and executes whatever commands are necessary to create the target**
 - **If no target is given then make creates the first target in the makefile**
- **There is typically one makefile per directory**

Useful Flags

- **make -f mymakefile.mk target**
 - make target using the file mymakefile.mk
- **make -n target**
 - don't really make just list the commands that would be executed if we were making
- **make -i target**
 - Ignore error codes
- **make -k target**
 - Similar to -i option except make goes on to make the non-error producing objects
- **make -q target**
 - Don't execute any commands merely determine if the target of the make is up to date
- **make -s target**
 - Suppress printing of the executed commands to the screen
- **make -j target**
 - Run multiple commands at once
- **make -t target**
 - Don't run any of the make file commands merely touch all of the target files bringing them up to date
- **make -d target**
 - Print information on what targets are being created and what commands are being run

Parallelism

- **make can run multiple commands at the same time using the -j option**
- **These is handy when your machine has multiple processors**
- **If we the program depends on 4 object files these would be compiled in parallel and the compiler would wait to link until all are compiled**

make and RCS

- **RCS allows a team of programmers to work on a set of programs together**
- **Programmers can store and check out source code in a controlled way**
- **Source files must be checked out before they can be used**
- **The date reflects their check out time not their modification time**
- **GNU make can find RCS files assuming that they reside in the same directory as our source files or in a subdirectory named RCS. These files are automatically checked out as part of the build**

- **Example**

make program.o

co RCS/program.c,v program.c

RCS/program.c,v --> program.c,v

revision 1.1

done

cc -c program.c -o program.o

rm program.c

- **We note that make cleans things up by removing the program.c file that it checked out**

Error Messages

- **No rule to make target 'target'. Stop.**
 - We have provided no specific rules in our make file to create this target.
 - Possible causes
 - Typo on target name
 - Perhaps we are in the wrong directory
- **"target" is up to date**
 - Self evident.
- **missing separator**
 - Missing tab at the beginning of a target line
 - Missing # in a comment statement

Error Messages - II

- **Target target not remade because of errors**
 - Only occurs when make is invoked with a -k
 - Implies that make bailed on this target
- **command: Command not found**
 - Probably a typo on a command
 - Command might be an alias known to the command prompt but not in the make file
- **illegal option -- option**
 - make has been invoked with a bad command line option
 - We note that some make options that support multiple characters require a - -
- **option requires an argument -- option**
 - make invoked with an option without the necessary parameter
 - eg) make -f
- **Recursive variable 'macro' references itself (eventually).**
Stop
 - Recursive macro references
 - CFLAGS = \$(CFLAGS)

Simple Example of a Multi-file Makefile

```
CC=gcc
```

```
CFLAGS=-g
```

```
OBJ=triangle_drive.o triangle.o
```

```
all: triangle_drive
```

```
triangle_drive: $(OBJ)
```

```
$(CC) $(CFLAGS) -o triangle_drive $(OBJ)
```

```
triangle_drive.o:triangle.h triangle_drive.c
```

```
triangle.o:triangle.c
```

Additional References

- **O'Reilly (Nutshell Handbook) “Managing Projects with make.”**

- **GNU on-line documentation**
 - **www.gnu.ai.mit.edu**