CSc 352: Elementary “make”

What is “make”?

- make is a Unix tool that simplifies the task of compiling large and complex software projects
  - the programmer creates a *Makefile* that speficies:
    - which files depend on which other files
    - what commands to run when a file changes and needs to be (re)compiled
  - *make* reads Makefile and runs commands as necessary
    - not such a big deal for small single-source-file programs
    - very helpful when a large number of files involved
    - can be used for many tasks that involve running some commands on files that have changed

Example

- Suppose we have four C programs:

source file	executable	compiler command
mayan.c	mayan	gcc –Wall mayan.c -o mayan
palindromes.c	palindromes	gcc –Wall palindromes.c –o palindromes
rotstr.c	rotstr	gcc –Wall rotstr.c –o rotstr
vowels.c	vowels	gcc –Wall vowels.c –o vowels

- We fix a bug in palindromes.c
  - need to recompile palindromes.c
  - but *not* the remaining files!
- We can do this using make

Example

Makefile

```
mayan : mayan.c
\t gcc –Wall mayan.c  –o mayan

palindromes : palindromes.c
\t gcc –Wall palindromes.c  –o palindromes

rotstr : rotstr.c
\t gcc –Wall rotstr.c  –o rotstr

vowels : vowels.c
\t gcc –Wall vowels.c  –o vowels
```

What it means

1. The file mayan is built from the file mayan.c
2. To build mayan from mayan.c, execute this command

(and similarly for the other files)

Makefile structure: rules

“target”

Makefile

```
mayan : mayan.c
    gcc –Wall mayan.c  –o mayan
```

file(s) the target depends on (≥ 0)
(“prerequisites”)

if any of these files is changed, the target must be recompiled.

“rule”

command (≥ 0) to execute if target is out of date w.r.t. the files it depends on

In general, a makefile will have many rules:
typically, one for each target

How make works

Makefile

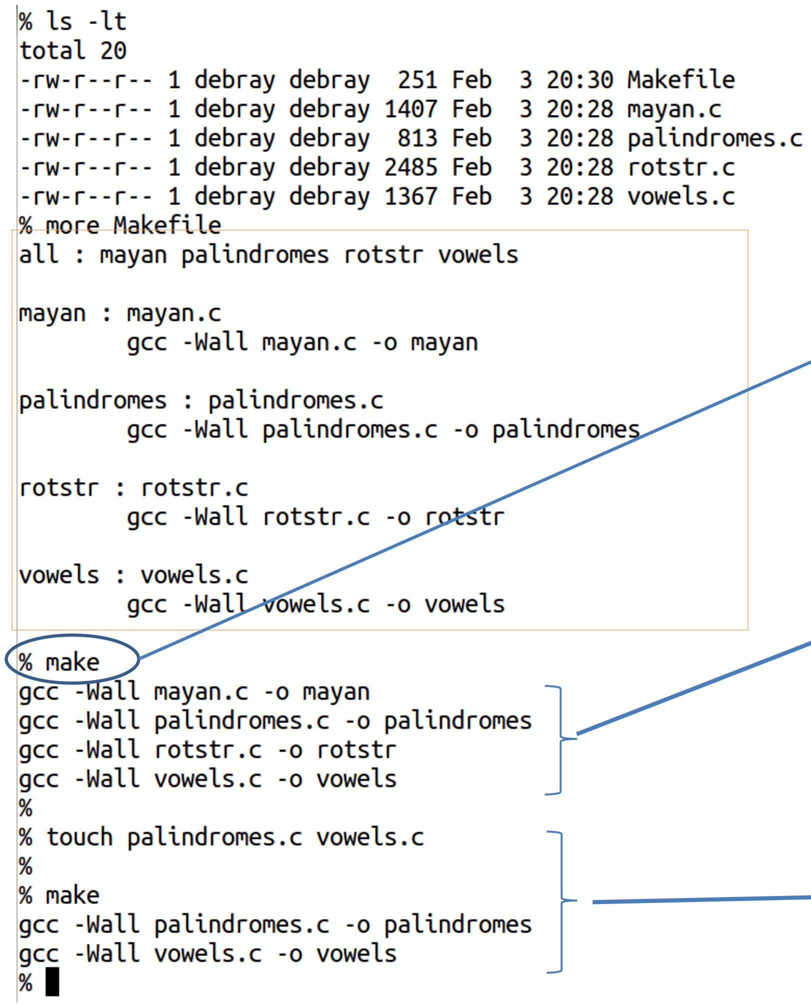
```
target : prerequisite_1  …  prerequisite_n
        command
```

Behavior of make

1. if target does not exist:
   – run command
2. else: if target is older than any of its prerequisites:
   – run command
3. else: /* (target is up to date) */
   – do nothing

If make is invoked without specifying a target, it uses the first target in the makefile by default

Example



```
% ls -lt
total 20
-rw-r--r-- 1 debray debray  251 Feb  3 20:30 Makefile
-rw-r--r-- 1 debray debray 1407 Feb  3 20:28 mayan.c
-rw-r--r-- 1 debray debray  813 Feb  3 20:28 palindromes.c
-rw-r--r-- 1 debray debray 2485 Feb  3 20:28 rotstr.c
-rw-r--r-- 1 debray debray 1367 Feb  3 20:28 vowels.c
% more Makefile
all : mayan palindromes rotstr vowels

mayan : mayan.c
        gcc -Wall mayan.c -o mayan

palindromes : palindromes.c
        gcc -Wall palindromes.c -o palindromes

rotstr : rotstr.c
        gcc -Wall rotstr.c -o rotstr

vowels : vowels.c
        gcc -Wall vowels.c -o vowels
% make
gcc -Wall mayan.c -o mayan
gcc -Wall palindromes.c -o palindromes
gcc -Wall rotstr.c -o rotstr
gcc -Wall vowels.c -o vowels
%
% touch palindromes.c vowels.c
%
% make
gcc -Wall palindromes.c -o palindromes
gcc -Wall vowels.c -o vowels
% █
```

Makefile

to run make, type the command
make

make prints out the commands it executes as it executes them

when make is re-run, only the commands for the updated prerequisites are executed

Creating a makefile

1. What are the targets?
   - figure out which files are *created* from other files and which need to be *re-created* when any of those files change.
2. For each target **foo**:
   - what are the files which, if changed, would require us to re-create **foo**?

     These are the *prerequisites* for **foo** (let's say **bar**$_1$ ... **bar**$_n$)
3. What commands do we use to (re-)create **foo**?
   - say: cmd$_1$ ... cmd$_m$

Creating a makefile

- The resulting rule for foo is:

foo: $\textbf{bar}_1$ $\textbf{bar}_2$ **...** $\textbf{bar}_n$
$\boxed{\text{tab}}$ cmd_1
$\boxed{\text{tab}}$ cmd_2
...
$\boxed{\text{tab}}$ cmd_m

or:

foo: $\textbf{bar}_1$ $\textbf{bar}_2$ **...** $\textbf{bar}_n$
$\boxed{\text{tab}}$ cmd_1 ; cmd_2 ; ... cmd_m

Another example

We can use make to build a program in different ways:

```
# 32-bit executables
mayan-32:
    gcc –m32  mayan.c  –o mayan


# 64-bit executables
mayan-64:
    gcc –m64  mayan.c –o mayan
```

```
# unoptimized
mayan:
    gcc   mayan.c    –o mayan


# optimized
mayan-opt:
    gcc –O3  mayan.c –o mayan
```

Phony Targets

Makefile

This rule has no command!

```
all : mayan palindromes rotstr

mayan : mayan.c
	gcc  -Wall  mayan.c -o mayan

palindromes : palindromes.c
	gcc  -Wall  palindromes.c -o palindromes

rotstr : rotstr.c
	gcc  -Wall  rotstr.c -o rotstr

clean:
	/bin/rm -f mayan palindromes rotstr
```

This rule has no prerequisites!

- We don't really intend to "build" the targets **all**, **clean**
  - Such "targets" are called *phony targets*
- The rules would not work if we accidentally introduced files named **all**, **clean**
  - to prevent this problem, declare them to be phony targets

Phony targets

```
.PHONY: all clean

all : mayan palindromes rotstr

mayan : mayan.c
    gcc  -Wall  mayan.c -o mayan

palindromes : palindromes.c
    gcc  -Wall  palindromes.c -o palindromes

rotstr : rotstr.c
    gcc  -Wall  rotstr.c -o rotstr

clean:
    /bin/rm -f mayan palindromes rotstr
```

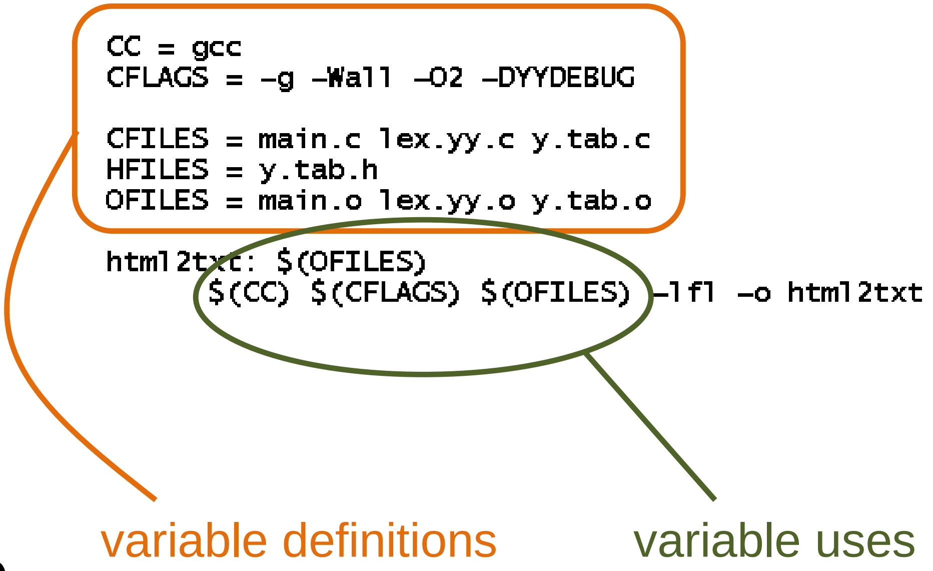
Variables

- A variable is a name that represents a text string (its *value*)
  - also called “macros”
- defined as:

  XYZ = value

- used as: $(XYZ)
- makes it easier to write, understand makefiles

Example: Part of a makefile from a CSC 453 assignment:



```
CC = gcc
CFLAGS = -g -Wall -O2 -DYYDEBUG

CFILES = main.c lex.yy.c y.tab.c
HFILES = y.tab.h
OFILES = main.o lex.yy.o y.tab.o

html2txt: $(OFILES)
	$(CC) $(CFLAGS) $(OFILES) -lfl -o html2txt
```

Summary of concepts

- Rules
  - represent dependencies between files
  - specified using targets, prerequisites, commands
- Phony targets
- Variables