make

UVic SEng 265

Daniel M. German
Department of Computer Science
University of Victoria

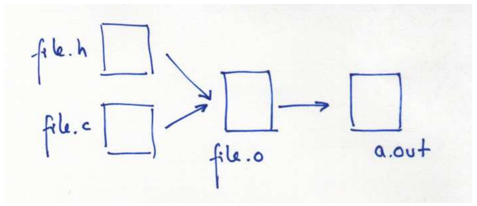
December 2, 2002 Version: 1.00

make

- Re-compiling larger programs takes much longer than re-compiling short programs.
- You only work in a small section of the code, so you don't want to recompile everything all the time
- Most of the code remains unchanged from compilation to compilation
- `make` recompiles only those files that need to be recompiled because of the changes you have made

A simple compilation

- Your program consists of `file.h, file.c`
- You compile `file.c`: `gcc file.c`
- The compiler generates first `file.o` and then `a.out`



Compiling with several files

- As your program grows you start to split the C file into smaller ones
- Example of compiling 2 source C files with one common include file (`green.c, blue.c, common.h`):

  `gcc green.c blue.c`
- The compiler compliles `green.c` and `blue.c` and then it links them together to create `a.out`

Compiling with several files ...

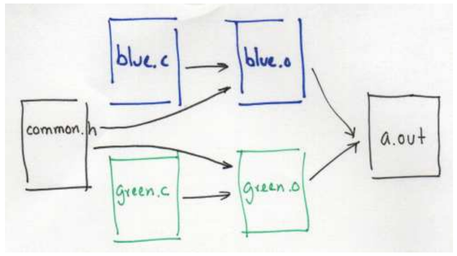
- We can also compile them one at a time:

```
gcc -c blue.c
gcc -c green.c
gcc blue.o green.o
```

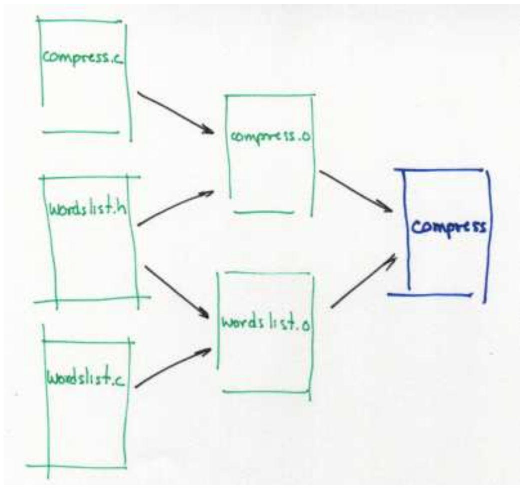
- In order to create `blue.o`, we need `blue.c` and `common.h`
- In order to create `green.o`, we need `green.c` and `common.h`
- In order to create `a.out`, we need `green.o` and `blue.o`

Dependencies

- Each generated file *depends* on others to be created.
- For example: `blue.o` depends on `blue.c` and `common.h`
- In general, each created file depends on at least one input file.
- This dependency relation can be depicted with a graph called "dependency graph"

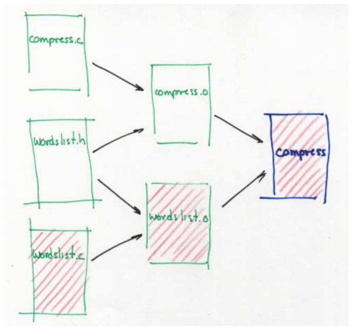


Dependency Graph for a program



How dependency works

- Suppose we change `wordslist.c`, then we only want to recompile this file and then recreate `compress`
- By following the edges of the graph, we quickly see which files need to be recreated



A simple Makefile

```
default: compress

compress: compress.o wordslist.o
        gcc -o compress compress.o wordslist.o

wordslist.o: wordslist.c wordslist.h
        gcc -c wordslist.c

compress.o: compress.c wordslist.h
        gcc -c compress.c
```

- By default, `make` reads its input from the file called Makefile
- This file contains a textual version of the dependency graph, including the command to run to generate the output for each edge of the graph

Translating the Dependency Graph

- The format for the creation of each node of the dependency graph is:

```
target: source-file(s)
        commands
```

- Don’t forget to proceed the command with a tab
- Example:

```
wordslist.o: wordslist.c wordslist.h
        gcc -c wordslist.c  # don’t forget the tab
```

- Comments start with # (perl style)

Variables

- You can use variables:

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

- If you want to include $ in your Makefile, write $$

Implicit Compilation

- Certain standard ways of remaking target files are used very often. For example, one customary way to make an object file is from a C source file using the C compiler, ‘gcc’.
- **Implicit rules** tell `make` how to use customary techniques so that you do not have to specify them in detail when you want to use them.
- For example, C compilation typically takes a ‘.c’ file and makes a ‘.o’ file.
- `make` ’ applies the implicit rule for C compilation when it sees this combination of file name endings.

Example of Using Implicit Rules

```
default: single

CFLAGS = -Wall -pedantic -ansi -g  -DNDEBUG
CC = gcc
LDLIBS  = -lm
INCLUDES =  debug.h

single: single.o teams.o input.o

single.o: teams.h single.c $(INCLUDES)

teams.o: teams.h teams.c input.h $(INCLUDES)

input.o: input.h input.c $(INCLUDES)

clean:
       rm -f *.o
```

Implicit Rules

- Compiling `.c:` into `.o:`
  `$(CC) -c $(CPPFLAGS) $(CFLAGS)`
- Linking a single `.o` into an executable:
  `$(CC) $(LDFLAGS) file.o $(LOADLIBES) $(LDLIBS)`

Compiling and linking several files

- In this case, the implicit rules above prevail, with some extras. Example: the rule `x:  y.o z.o` will generate the following commands:

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

Rules that do not create targets

- Sometimes we want to execute a command over and over again
- If you write a rule whose commands will not create the target file, the commands will be executed every time the target comes up for remaking. Here is an example:

```
clean:
        rm *.o temp
```

Using make for more than just programming

```
FILE = 13_make
default: $(FILE).pdf $(FILE)_4up.pdf
%.dvi: %.tex
        latex $<

%.ps: %.dvi
        dvips -t letter -t landscape -o $@ $<

$(FILE)_4up.ps: $(FILE).ps
        psnup -r -pletter -4 $< $@

$(FILE)_4up.pdf: $(FILE)_4up.ps
        ps2pdf $< $@

$(FILE).pdf: $(FILE).ps
        ps2pdf $< $@

pdfs: $(FILE).pdf $(FILE)_4up.pdf

copy_pdfs:
        cp *.pdf ../../html/lectures
```