Make

- Motivation
- Separable compilation & dependencies
- Expressing dependencies textually
- Rules
- Examples beyond programming

“make” and Makefiles

- Large software projects usually consist of dozen (perhaps hundreds) of files
- Most of the files correspond to:
  - source code
  - object code
  - interface descriptions
  - configuration information
  - automatically-generated documentation
- A software **build**:
  - Constructed executable version of the program
- How do we build software efficiently?
  - a small change to one part of the program should not require the reprocessing of every other file

make & makefile(s)

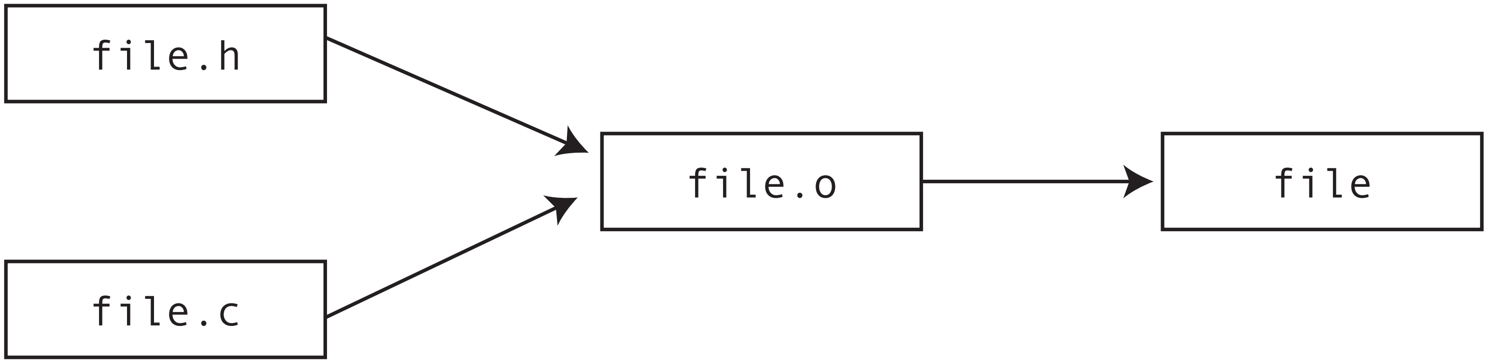
- **make** is an important programming utility
- It uses a **makefile**
  - This describes **dependencies**
  - Its format is very specific
- A **dependency** represents/encodes the some relationship between files in a project
  - if file A uses the information in file B…
  - and if file C does not use information in file B…
  - then any change to B should result in only A being reprocessed

make & makefile(s)

- **Insight: reprocessing several files instead of all project files can produce a real time savings**
  - **processing** often means **compilation**
  - But it can also mean re-generating files, relinking object code, re-running tests, etc.
  - However, it may be that most of the code remains unchanged from compilation to compilation
- Our interfile dependencies described within the **makefile** are used to determine what is to be re-processed
  - Guided by the dependencies, **make** directs recompile (etc.) for only those files that need re-processing because of changes

A simple compilation

- Your program consists of file.h, file.c
- You compile file.c: gcc file.c
- The compiler generates first file.o then a.out if we do not specify differently.
- (We normally write gcc file.c –o file to change the name of the executable file)



Compiling with several files

- Good programming practice suggests we break programs into smaller modules
- Each module corresponds to a separate file
- Example:
  - compiling two source C files with a common include file (`red.c, yellow.c, common.h`)
    `gcc red.c yellow.c`
  - the compiler translates `red.c` and `yellow.c` into object files, and then creates an executable named `a.out`
  - preferred is `gcc red.c yellow.c -o colour` to create executable named `colour`

Compiling with separate files...

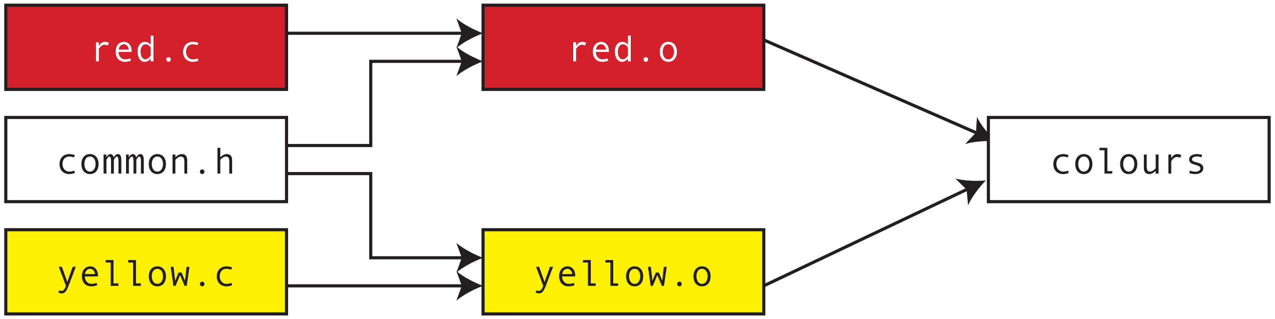
- We can also compile each source-code file, one at a time:

```
gcc -c red.c
gcc -c yellow.c
gcc red.o yellow.o -o colours
```

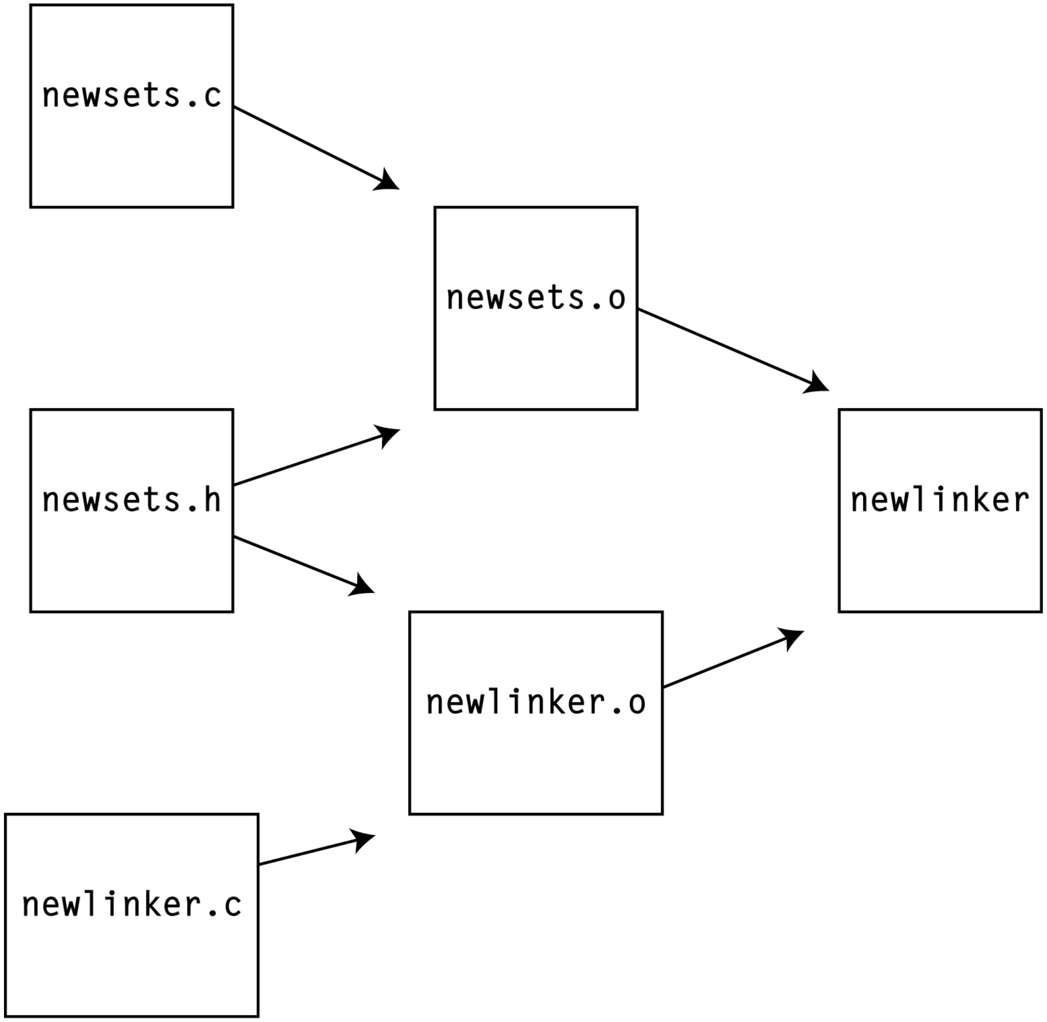
- In order to create `red.o`, we need `red.c` and `common.h`
- In order to create `yellow.o`, we need `yellow.c` and `common.h`
- In order to create `colours`, we need `red.o` and `yellow.o`

Dependencies

- Each generated file depends on others to be created.
- For example: `red.o` depends on `red.c` and `common.h`
- In general, each created file depends on at least one input file.
- This dependency relationship can be drawn as a graph called a "dependency graph"

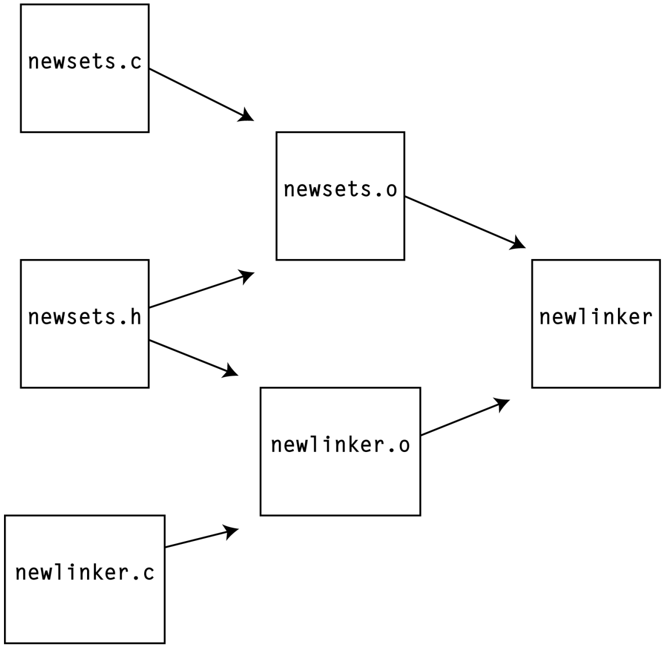


Dependency graph for a program



makefile: example

```
SHELL=/usr/bin/bash

CC=gcc

newlinker :   newlinker.o newsets.o
    $(CC) -o newlinker newlinker.o newsets.o

newlinker.o:  newlinker.c newsets.h
    $(CC) -c -g -Wall -std=c99 newlinker.c

newsets.o: newsets.c newsets.h
    $(CC) -c -g -Wall -std=c99 newsets.c

clean:
    -rm newlinker.exe newlinker.o newsets.o
```



using make: example

- using makefile from previous page

```
> make
gcc -c -g -Wall -std=c99 newlinker.c
gcc -c -g -Wall -std=c99 newsets.c
gcc -o newlinker newlinker.o newsets.o

> touch newsets.c

> make
gcc -c -g -Wall -std=c99 newsets.c
gcc -o newlinker newlinker.o newsets.o
```

- typing **make** with no arguments means **use first rule in the makefile**

makefile: features

- rules: consists of three parts
  - **target**: some name
    - could be the name of a program
    - could be a name we give to a set of programs
  - **dependencies**: list of files (and possibly empty)
  - **command**: UNIX command needed to perform work for target
  - **always put a tab (a <u>real</u> tab!) before the list of commands!.**
  - comments are shell-style (lines beginning with "#" character)
- variables
  - clears up redundancy / repetition within a makefile
  - eases the modification of makefiles
  - defined on their own line
  - used with a combination of $ and ()
  - if you wish to refer to '$' in the makefile, call it $$.

More about variables

- Another example

```
OBJECTS=data.o main.o io.o

project1: $(OBJECTS)
	gcc $(OBJECTS) -o project1

data.o: data.c data.h
	gcc -c data.c

main.o: data.h io.h main.c
	gcc -c main.c

io.o: io.h io.c
	gcc -c io.c
```

Implicit compilation

- Certain standard ways of remaking target files are often used. 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 using implicit rules

- Compiling .c: into .o:

```
$(CC) file.c -c $(CPPFLAGS) $(CFLAGS)
```

- Linking a single .o into an executable:

```
$(CC) $(LDFLAGS) file.o $(LOADLIBS) $(LDLIBS)
```

Example using implicit rules

```
default: single

CFLAGS=-Wall -pedantic -std=c99 -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
```

Automatic variables

- These may appear in the explicit rule of a target
  - Their benefit is to exploit rule patterns in a way that reduces error
- There are four of interest to us

$@	the target of the rule
$^	all the dependencies of the rule
$<	the first dependency of the rule
$?	all dependencies "newer" than the target

- Several others also exist
  - There is even pattern matching in "make"...
  - ... but we'll avoid that for now

using automatic variables

```
SHELL=/usr/bin/bash

CC=gcc


# the rule was:
#   $(CC) -o newlinker newlinker.o newsets.o
#
# but now we've modified it.

newlinker :   newlinker.o newsets.o
    $(CC) -o $@ $^

newlinker.o:  newlinker.c newsets.h
    $(CC) -c $<

newsets.o: newsets.c newsets.h
    $(CC) -c $<

clean:
    -rm newlinker.exe newlinker.o newsets.o
```

makefile: another example

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        gcc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        gcc -c main.c
kbd.o : kbd.c defs.h command.h
        gcc -c kbd.c
command.o : command.c defs.h command.h
        gcc -c command.c
display.o : display.c defs.h buffer.h
        gcc -c display.c
insert.o : insert.c defs.h buffer.h
        gcc -c insert.c
search.o : search.c defs.h buffer.h
        gcc -c search.c
files.o : files.c defs.h buffer.h command.h
        gcc -c files.c
utils.o : utils.c defs.h
        gcc -c utils.c
```

Make is for more than 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
```

additional features

- **phony targets**
  - correspond to actions taken which depend on no files
  - “clean”: often used to delete object files from a set of subdirectories
- **recursive makefiles**
  - Gnu’s **gmake** and Microsoft’s **nmake**
  - rather than construct one large makefile, smaller makefiles are kept in each sub-directory
  - makefile in top-most directory is used to launch builds based on sub-directory makefiles
- **include files**
  - the same information (e.g., variable values) may be needed in separate makefiles
  - write this information once, and then write an “include” statement in the appropriate makefiles

Who writes “makefiles”?

- for small projects:
  - you
  - course instructor
  - project administrator
- for larger projects:
  - tools for discovering dependencies (autotools)
  - configuration programs which construct makefiles for specific environment
- makefile “gotchas”
  - **use “tab” character to indent commands!!!**
  - the “\” is used to continue commands on another line

Beyond make

- Cmake
  - two-step process (build-environment setup + build)
  - Mostly seen with C++ projects
- ant + itch:
  - a little bit like "make" but for Java
  - "procedural" like make
- maven:
  - "declarative" (i.e.,uses conventions to determine what needs to be built)
  - makes possible a much more complex build chain, including downloading / install code from other servers
- many, many, many others
- Continuous integration tools
  - Jenkins