COS 217: Introduction to Programming Systems

Building Multifile Programs with make

PRINCETON UNIVERSITY

1

Multi-File Programs

intmath.h (interface)

```
#ifndef INTMATH_INCLUDED
#define INTMATH_INCLUDED
int gcd(int i, int j);
int lcm(int i, int j);
#endif
```

intmath.c (implementation)

```
#include "intmath.h"

int gcd(int i, int j)
{  int temp;
   while (j != 0)
   { temp = i % j;
     i = j;
     j = temp;
   }
   return i;
}

int lcm(int i, int j)
{  return (i / gcd(i, j)) * j;
}
```

testintmath.c (client)

```
#include "intmath.h"
#include <stdio.h>

int main(void)
{  int i;
   int j;
   printf("Enter the first integer:\n");
   scanf("%d", &i);
   printf("Enter the second integer:\n");
   scanf("%d", &j);
   printf("Greatest common divisor: %d.\n",
      gcd(i, j));
   printf("Least common multiple: %d.\n",
      lcm(i, j));
   return 0;
}
```

Note: intmath.h is #included into intmath.c and testintmath.c

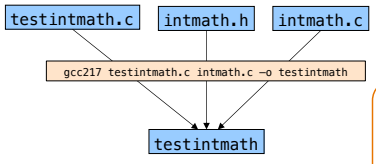
3

3

Motivation for Make (Part 1)

Building `testintmath`, approach 1:

- Use one `gcc217` command to preprocess, compile, assemble, and link



4

4

https://xkcd.com/303/

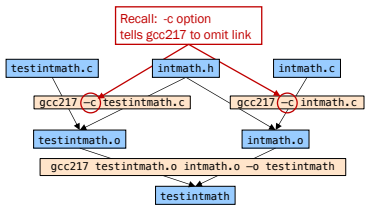
5

5

Motivation for Make (Part 2)

Building `testintmath`, approach 2:

- Preprocess, compile, assemble to produce .o files
- Link to produce executable binary file



6

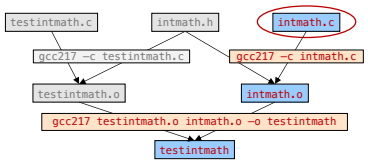
6

Partial Builds

Approach 2 allows for **partial builds**

- Example: Change `intmath.c`
  - Must rebuild `intmath.o` and `testintmath`
  - No need to rebuild `testintmath.o`

If program contains many files, could save hours of build time



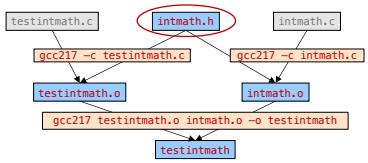
7

7

Partial Builds

However, changing a .h file can be more dramatic

- Example: Change `intmath.h`
  - `intmath.h` is `#include`'d into `testintmath.c` and `intmath.c`
  - Must rebuild `testintmath.o`, `intmath.o`, and `testintmath`



8

8

Wouldn't It Be Nice If...

Observation

- Doing partial builds manually is tedious and error-prone
- Wouldn't it be nice if there were a tool...

How would the tool work?

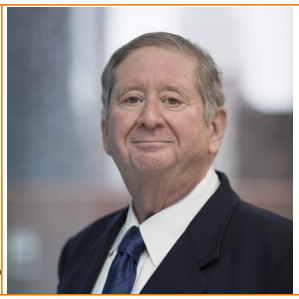
- Input:
  - Dependency graph (as shown previously)
    - Specifies file dependencies
    - Specifies commands to build each file from its dependents
  - Date/time stamps of files
- Algorithm:
  - ***If*** file B depends on A ***and*** date/time stamp of A is newer than date/time stamp of B, ***then*** rebuild B using the specified command

That's `make`!

9

9

Obligatory Princeton Context



Stuart Feldman '68

- Chief Scientist at Schmidt Futures
- Former President of ACM
- AAAS, IEEE, and ACM fellow
- Board Chair of CMD-IT

Created `make` at Bell Labs in 1976

10

10

Make Command Syntax

Command syntax

```
$ man make
SYNOPSIS
    make [-f makefile] [options] [targets]
```

- `makefile`
  - Textual representation of dependency graph
  - Contains **dependency rules**
  - Default name is `makefile`, then `Makefile`
- `target`
  - What `make` should build
  - Usually: .o file or executable binary file
  - Default is first one defined in `makefile`

12

12

Dependency Rules in Makefile

Dependency rule syntax

```
target: prerequisites
    <tab>command
```

- `target`: the file you want to build
- `dependencies (aka prerequisites)`: the files needed to build the target
- `command`: what to execute to build the target

Dependency rule semantics

- Build `target` if it doesn't exist
- Rebuild `target` iff it is older than any of its `dependencies`
- Use `command` to do the build

13

13

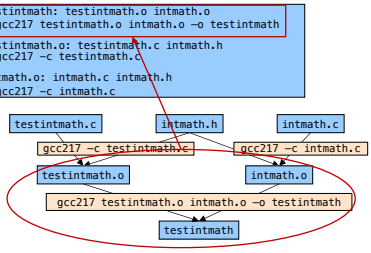
Makefile Version 1

Makefile

```
testintmath: testintmath.o intmath.o
  gcc217 testintmath.o intmath.o -o testintmath

testintmath.o: testintmath.c intmath.h
  gcc217 -c testintmath.c

intmath.o: intmath.c intmath.h
  gcc217 -c intmath.c
```



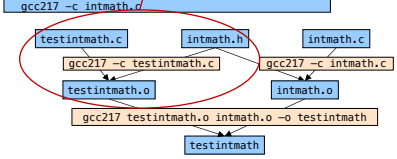
14

14

Makefile Version 1

Makefile

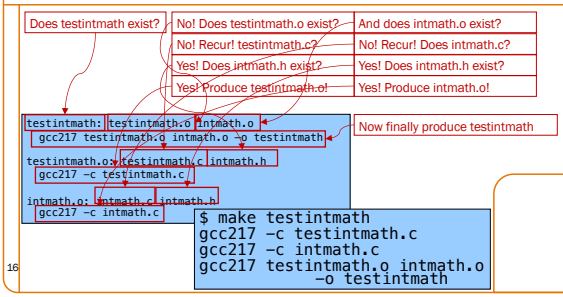
```
testintmath: testintmath.o intmath.o
  gcc217 testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
  gcc217 -c testintmath.c
intmath.o: intmath.c intmath.h
  gcc217 -c intmath.c
```



15

15

Version 1 in Action




```
$ make testintmath
gcc217 -c testintmath.c
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o
       -o testintmath
```

16

16

Version 1 in Action

At first, to build testintmath make issues all three gcc commands

Use the touch command to change the date/time stamp of intmath.c

```
$ make testintmath
gcc217 -c testintmath.c
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath
$ touch intmath.c
$ make testintmath
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath
$ make testintmath
make: `testintmath' is up to date.
$ make
make: `testintmath' is up to date.
```

make does a partial build

make notes that the specified target is up to date

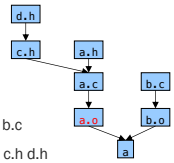
The default target is testintmath, the target of the first dependency rule

17

17

make up your mind

Q: If you were making a `Makefile` for this program, what should `a.o` depend on?



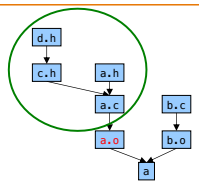
A. a
B. a.c
C. a.c b.c
D. a.h c.h d.h
E. a.c a.h c.h d.h

E: its own .c file, and all .h files (directly or indirectly) included.

18

18

Makefile Guidelines



a.o: a.c a.h c.h d.h

In a proper Makefile, each object file:

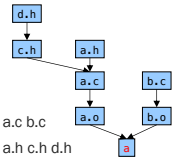
- Depends upon its .c file
  - Does not depend upon any other .c file
  - Does not depend upon any .o file
- Depends upon any .h files that are #included **directly or indirectly**

19

19

building understanding

Q: If you were making a `Makefile` for this program, what should a depend on?



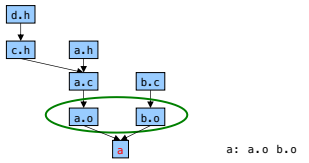
A. a.o b.o
B. a.o b.o a.c b.c
C. a.o b.o a.h c.h d.h
D. a.c b.c a.h c.h d.h
E. a.o b.o a.c b.c a.h c.h d.h

A: only .o files (the .o files' rules themselves will have .c and .h file dependencies, so these are covered recursively)

20

20

Makefile Guidelines



In a proper Makefile, each executable:

- Depends upon the .o files that comprise it
- Does not depend upon any .c files
- Does not depend upon any .h files

21

21

Non-File Targets (aka "pseudotargets")

Adding useful shortcuts for the programmer

- `make all`: create the final executable binary file(s)
- `make clean`: delete all .o files, executable binary file(s)
- `make clobber`: delete all Emacs backup files, all .o files, executable(s)

Commands in the example

- `rm -f`: remove files without querying the user
- Files ending in '~' and starting/ending in '#' are Emacs special files

```
all: testintmath
clobber: clean
  rm -f *~ \#*\#
clean:
  rm -f testintmath *.o
```

23

23

Makefile Version 2

```
# Dependency rules for non-file targets
all: testintmath
clobber: clean
  rm -f *~ \#*\#
clean:
  rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
  gcc217 testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
  gcc217 -c testintmath.c
intmath.o: intmath.c intmath.h
  gcc217 -c intmath.c
```

24

24

Version 2 in Action

make observes that "clean" target doesn't exist; attempts to build it by issuing "rm" command

Same idea here, but "clobber" depends upon "clean"

```
$ make clean
rm -f testintmath *.o

$ make clobber
rm -f testintmath *.o
rm -f *~ \#*\#

$ make all
gcc217 -c testintmath.c
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath

$ make
make: Nothing to be done for `all'.
```

"all" depends upon "testintmath"

"all" is the default target

25

25

Macros

`make` has a macro facility

- Performs textual substitution
- Similar to C preprocessor's `#define`

Macro definition syntax

```
macroname = macrodefinition
```

- `make` replaces `$(macroname)` with `macrodefinition` in remainder of Makefile

Example: Make it easy to change (or swap) build commands

```
CC = gcc217#m
YACC = bison -d -y
#YACC = yacc -d
```

Example: Make it easy to change build flags

```
CFLAGS = -D NDEBUG -O
```

27

27

Makefile Version 3

```
# Macros
CC = gcc217
# CC = gcc217m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
  rm -f *~ \#*\#
clean:
  rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
  $(CC) $(CFLAGS) testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
  $(CC) $(CFLAGS) -c testintmath.c
intmath.o: intmath.c intmath.h
  $(CC) $(CFLAGS) -c intmath.c
```

28

28

Makefile Gotchas

Beware:

- Each command (i.e., second line of each dependency rule) must begin with a tab character (ASCII `0x09`), not spaces
- Use the `rm -f` command with caution
- (More generally, be careful about automatically doing anything you can't undo!)
- Have something sensible as your default command

30

30

Making Makefiles

In this course
- Create Makefiles manually
- Perhaps start from Makefiles from this lecture?

Beyond this course
- Can use tools to generate Makefiles
  - See `mkmf`, others
- Copy-paste-edit forever!

31

31

Advanced: Automatic Variables

`make` has wildcard matching for generalizing rules
- `make` has "pattern" rules that use % in targets and dependencies
- `make` has variables to fill in the "pattern" in commands
  - `$@` : the target of the rule that was triggered
  - `$<` : the first dependency of the rule
  - `$?` : all the dependencies that are newer than the target
  - `$^` : all the dependencies

Examples:

```
testintmath: testintmath.o intmath.o
    $(CC) $(CFLAGS) $^ -o $@
%.o: %.c intmath.h
    $(CC) $(CFLAGS) -c $<
```

Not required (and potentially confusing!), but common.

32

32

Advanced: Implicit Rules

`make` has implicit rules for compiling and linking C programs
- `make` knows how to build x.o from x.c
  - Automatically uses $(CC) and $(CFLAGS)
- `make` knows how to build an executable from .o files
  - Automatically uses $(CC)

`make` has implicit rules for inferring dependencies
- `make` will assume that x.o depends upon x.c

Not required (and potentially confusing):
we'll never ask you to write these!

33

33

Makefile Version 4 & 5

```
testintmath.o: testintmath.c intmath.h
    $(CC) $(CFLAGS) -c intmath.c
testintmath.o: testintmath.c intmath.h
testintmath.o: intmath.h
```

```
intmath.o: intmath.c intmath.h
    $(CC) $(CFLAGS) -c intmath.c
intmath.o: intmath.c intmath.h
intmath.o: intmath.h
```

```
testintmath: testintmath.o intmath.o
    $(CC) testintmath.o intmath.o -o testintmath
testintmath: testintmath.o intmath.o
```

```
# Macros
CC = gcc217
CFLAGS =

# Dependency rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#*\#
clean:
    rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
testintmath.o: testintmath.c intmath.h
intmath.o: intmath.c intmath.h
```

Not required
(and potentially confusing)

34

34

Makefile Gotchas

Beware:

- To use an implicit rule to make an *executable*, the executable must have the same name as one of the .o files

Correct: `myprog: myprog.o someotherfile.o` ✓

Won't work: `myprog: somefile.o someotherfile.o` ✗

35

35

Make Resources

C Programming: A Modern Approach (King) Section 15.4

GNU make

- http://www.gnu.org/software/make/manual/make.html

37

37