



## Building



Fall 2017

1

## Goals of this Lecture



Help you learn about:

- The build process for multi-file programs
- Partial builds of multi-file programs
- **make**, a popular tool for automating (partial) builds

Why?

- A complete build of a large multifile program typically consumes many hours
- To save build time, software engineers do partial builds
- We automate (partial) builds using **make**

2

## Review: 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>

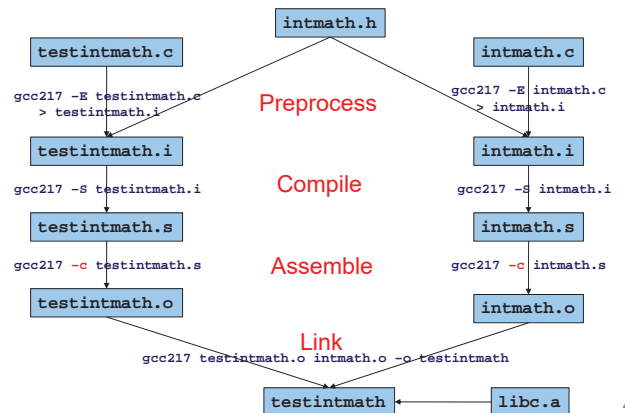
int main(void)
{
    int i;
    int j;
    printf("Enter the first integer:\n");
    scanf("%d", &i);
    printf("Enter the second integer:\n");
    scanf("%d", &j);
    printf("Greatest common divisor: %d.\n",
        gcd(i, j));
    printf("Least common multiple: %d.\n",
        lcm(i, j));
    return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

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

See precept handouts for stylistically better version

3

## Review: Multi-File Programs



4

## Agenda



### Motivation for Make

Make Fundamentals

Nonfile Targets

Macros

Abbreviations

Pattern Rules

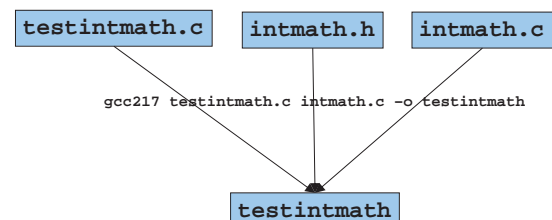
5

## Motivation for Make (Part 1)



Building testintmath, approach 1:

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



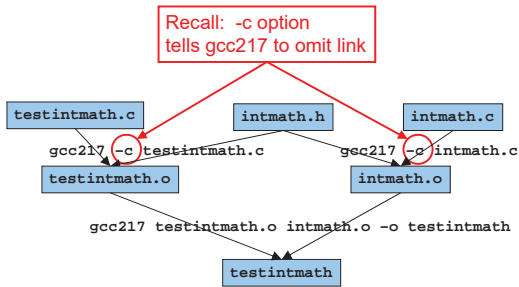
6

## Motivation for Make (Part 2)



### Building `testintmath`, approach 2:

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



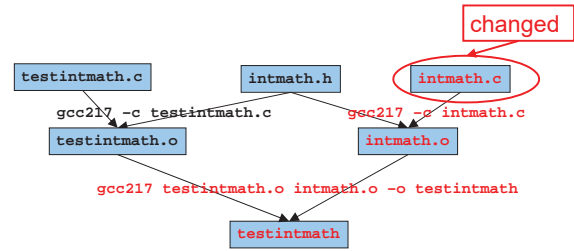
7

## Partial Builds



### Approach 2 allows for **partial builds**

- Example: Change `intmath.c`
  - Must rebuild `intmath.o` and `testintmath`
  - Need not rebuild `testintmath.o`!!!



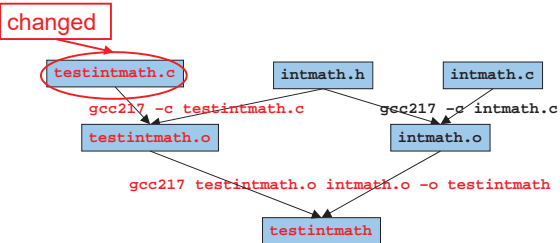
8

## Partial Builds



- Example: Change `testintmath.c`
  - Must rebuild `testintmath.o` and `testintmath`
  - Need not rebuild `intmath.o`!!!

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



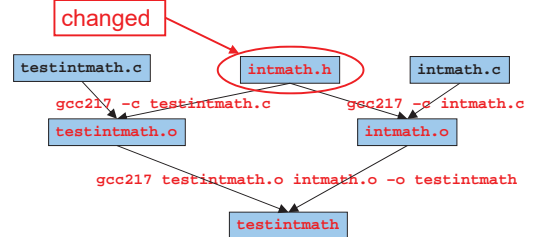
9

## Partial Builds



However, changing a `.h` file can be more dramatic

- Example: Change `intmath.h`
  - `intmath.h` is `#included` into `testintmath.c` and `intmath.c`
  - Changing `intmath.h` effectively changes `testintmath.c` and `intmath.c`
- Must rebuild `testintmath.o`, `intmath.o`, and `testintmath`



10

## Wouldn't It Be Nice...



### 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**!

11

## Agenda



Motivation for Make

**Make Fundamentals**

Nonfile Targets

Macros

Abbreviations

Pattern Rules

12

## The Make Tool



**Who?** Stuart Feldman '68

**When?** 1976

**Where?** Bell Labs

**Why?** Automate partial builds



(This is Stu Feldman recently;  
in 1976 he looked younger)

13

## Make Command Syntax



### Command syntax

```
make [-f makefile] [target]
```

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

14

## Dependency Rules



### Dependency rule syntax

```
target: dependencies  
    <tab>command
```

- **target**: the file you want to build
- **dependencies**: the files on which the target depends
- **command**: what to execute to create the target (after a TAB character)

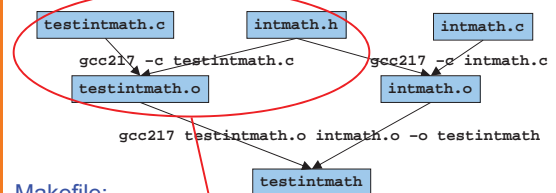
### Dependency rule semantics

- Build **target** iff it is older than any of its **dependencies**
- Use **command** to do the build

Work recursively; examples illustrate...

15

## 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
```

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

## Agenda



Motivation for Make

Make Fundamentals

**Nonfile Targets**

Macros

Abbreviations

Pattern Rules

18

## Non-File Targets



Adding useful shortcuts for the programmer

- **make all**: create the final executable binary file
- **make clean**: delete all .o files, executable binary file
- **make clobber**: delete all Emacs backup files, all .o files, executable binary file

Commands in the example

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

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

19

## Makefile Version 2



```
# Dependency rules for nonfile 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
```

20

## 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

21

## Agenda



Motivation for Make  
Make Fundamentals  
Nonfile Targets  
**Macros**  
Abbreviations  
Pattern Rules

22

## 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 build commands

**CC** = gcc217

Example: Make it easy to change build flags

**CFLAGS** = -D NDEBUG -O

23

## Makefile Version 3



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

# Dependency rules for nonfile 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
```

24

## Version 3 in Action



Same as Version 2

25

## Agenda



Motivation for Make  
Make Fundamentals  
Nonfile Targets  
Macros  
**Abbreviations**  
Pattern Rules

26

## Abbreviations



### Abbreviations

- Target file: **\$@**
- First item in the dependency list: **\$<**

### Example

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



```
testintmath: testintmath.o intmath.o
$(CC) $(CFLAGS) $< intmath.o -o $@
```

27

## Makefile Version 4



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

28

## Version 4 in Action



Same as Version 2

29

## Agenda



Motivation for Make  
Make Fundamentals  
Non-File Targets  
Macros  
Abbreviations  
**Pattern Rules**

30

## Pattern Rules



### Pattern rule

- Wildcard version of dependency rule
- Example:

```
%.o: %.c
$(CC) $(CFLAGS) -c $<
```

- Translation: To build a .o file from a .c file of the same name, use the command `$(CC) $(CFLAGS) -c $<`
- With pattern rule, dependency rules become simpler:

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

Can omit build command

31

## Pattern Rules Bonus



### Bonus with pattern rules

- First dependency is assumed

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



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

Can omit first dependency

32

## Makefile Version 5



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

# Pattern rule
%.o: %.c
$(CC) $(CFLAGS) -c $<

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

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

33

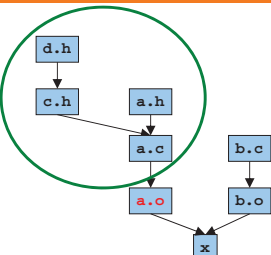
## Version 5 in Action



Same as Version 2

34

## Makefile Guidelines



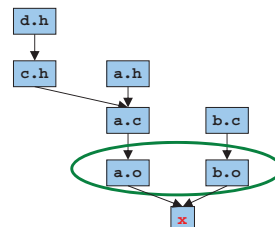
```
a.o: a.c a.h c.h d.h
gcc217 -c a.c
```

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 file that its .c file #includes **directly or indirectly**

35

## Makefile Guidelines



```
x: a.o b.o
gcc217 a.o b.o -o x
```

In a proper Makefile, each executable binary file:

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

36

## Making Makefiles



### In this course

- Create Makefiles manually

### Beyond this course

- Can use tools to generate Makefiles
  - See `mkmf`, others

37

## Makefile Gotchas



### Beware:

- Each command (i.e., second line of each dependency rule) must begin with a tab character, not spaces
- Use the `rm -f` command with caution

38

## Make Resources



*C Programming: A Modern Approach* (King) Section 15.4

### GNU make

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

39

## Summary



### Motivation for Make

- Automation of partial builds

### Make fundamentals (Makefile version 1)

- Dependency rules, targets, dependencies, commands

### Nonfile targets (Makefile version 2)

### Macros (Makefile version 3)

### Abbreviations (Makefile version 4)

### Pattern rules (Makefile version 5)

40