



# Building





# “Programming in the Large” Steps

## Design & Implement

- Program & programming style
- Common data structures and algorithms
- Modularity
- Building techniques & tools **<-- we are here**

## Debug

- Debugging techniques & tools

## Test

- Testing techniques **(done)**

## Maintain

- Performance improvement techniques & tools



# 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 multi-file program typically consumes many hours
- To save build time, a power programmer knows how to do partial builds
- A power programmer knows how to automate (partial) builds using **make**



# 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>

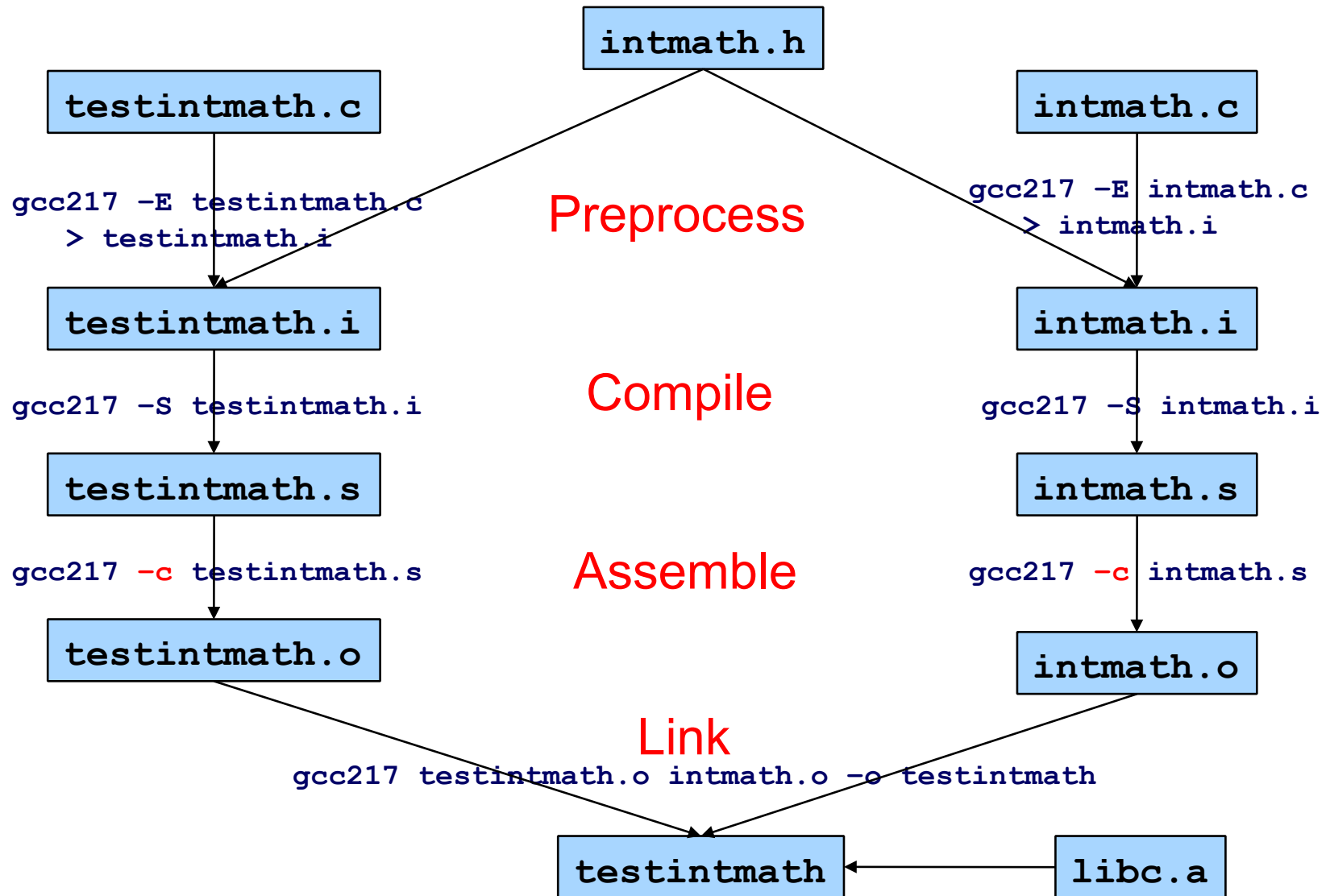
int main(void)
{ int i;
  int j;
  printf("Enter the first integer:\n");
  scanf("%d", &i);
  printf("Enter the second integer:\n");
  scanf("%d", &j);
  printf("Greatest common divisor: %d.\n",
        gcd(i, j));
  printf("Least common multiple: %d.\n",
        lcm(i, j));
  return 0;
}
```

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

See precept handouts for stylistically better version



# Review: Multi-File Programs





# Agenda

## **Motivation for Make**

Make Fundamentals

Non-File Targets

Macros

Abbreviations (if time)

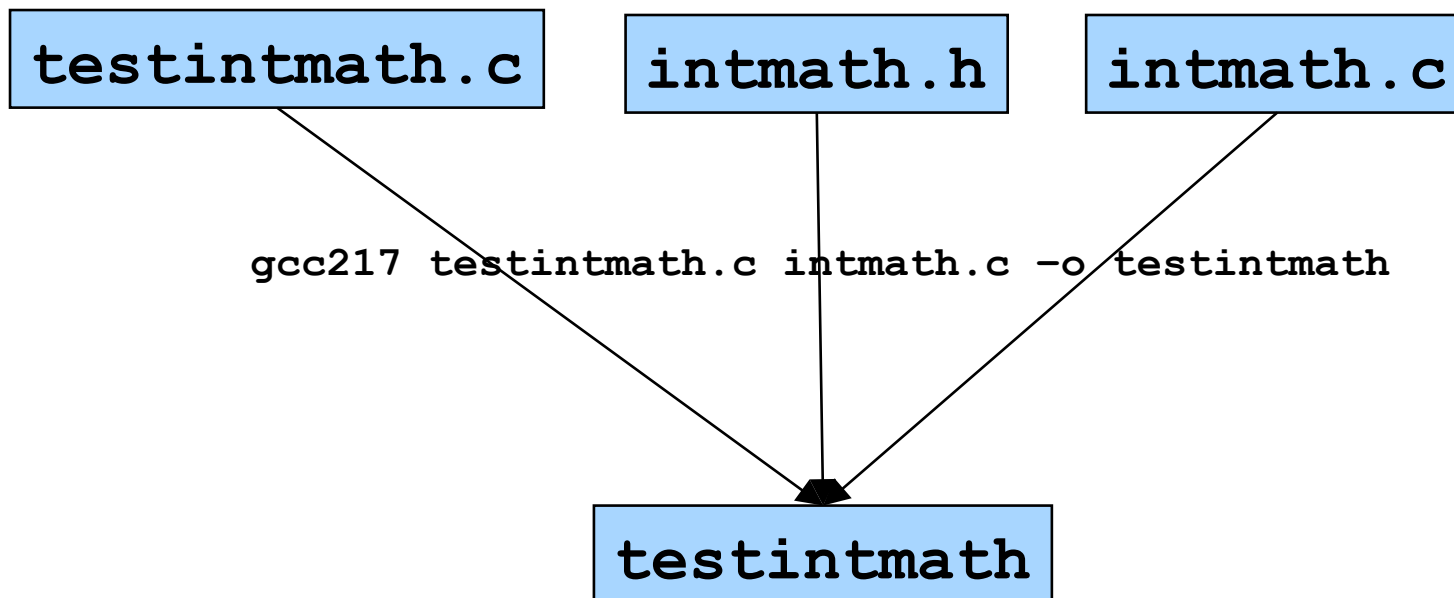
Pattern Rules (if time)



# Motivation for Make (Part 1)

## Building `testintmath`, approach 1:

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

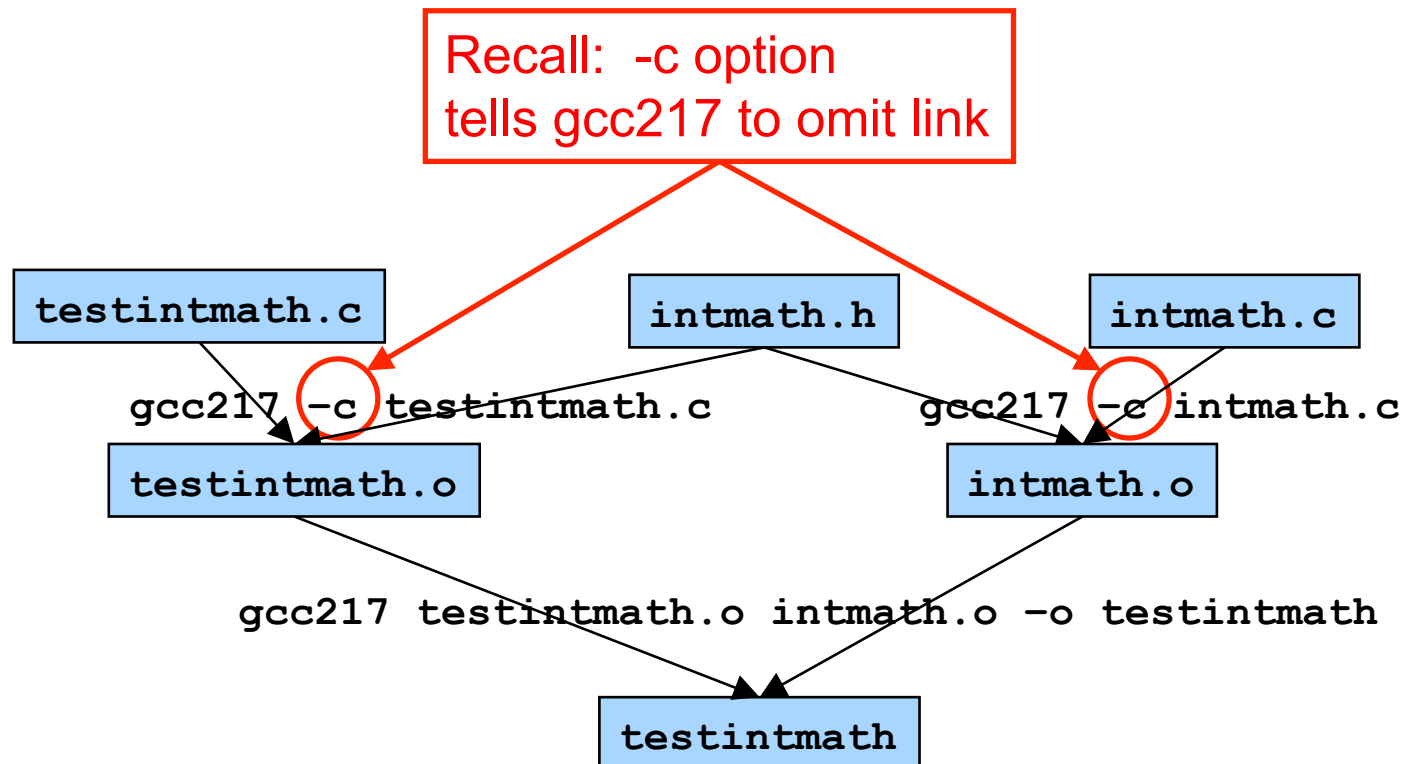




# Motivation for Make (Part 2)

## Building `testintmath`, approach 2:

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



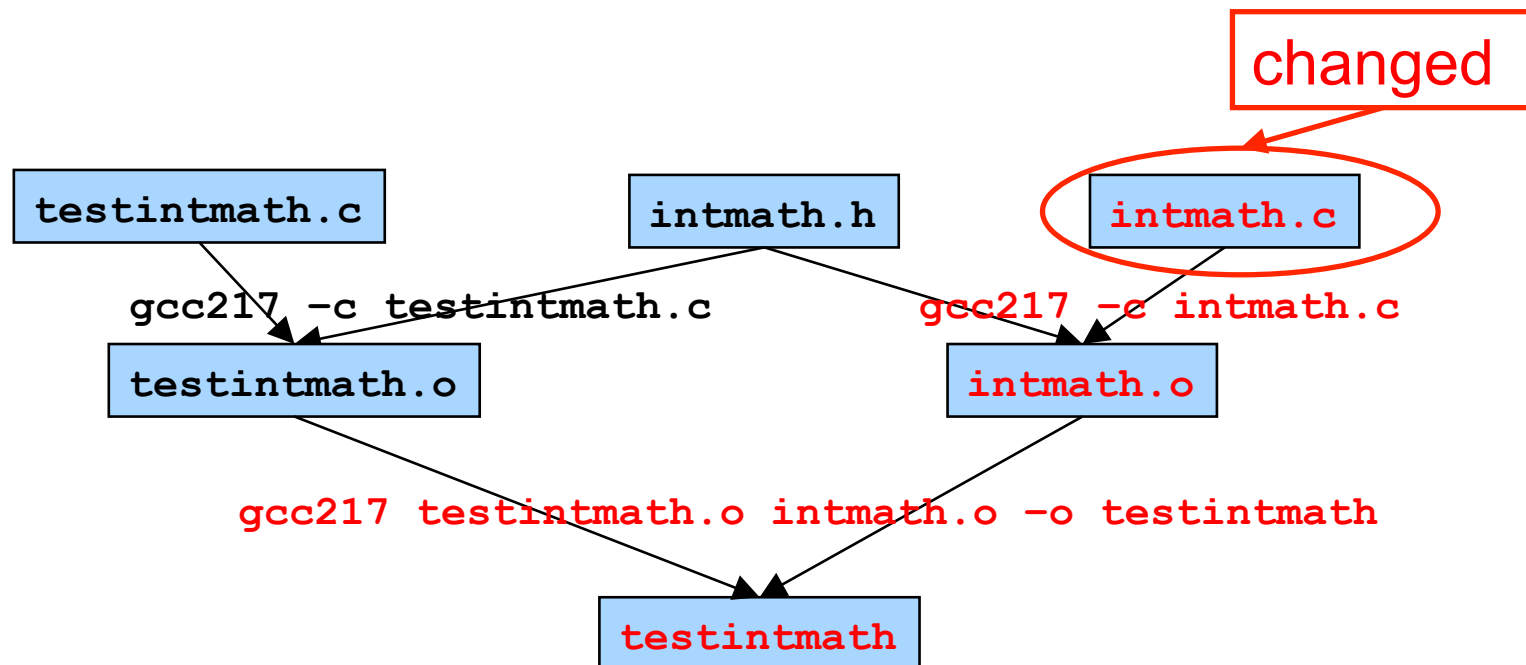




# Partial Builds

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

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

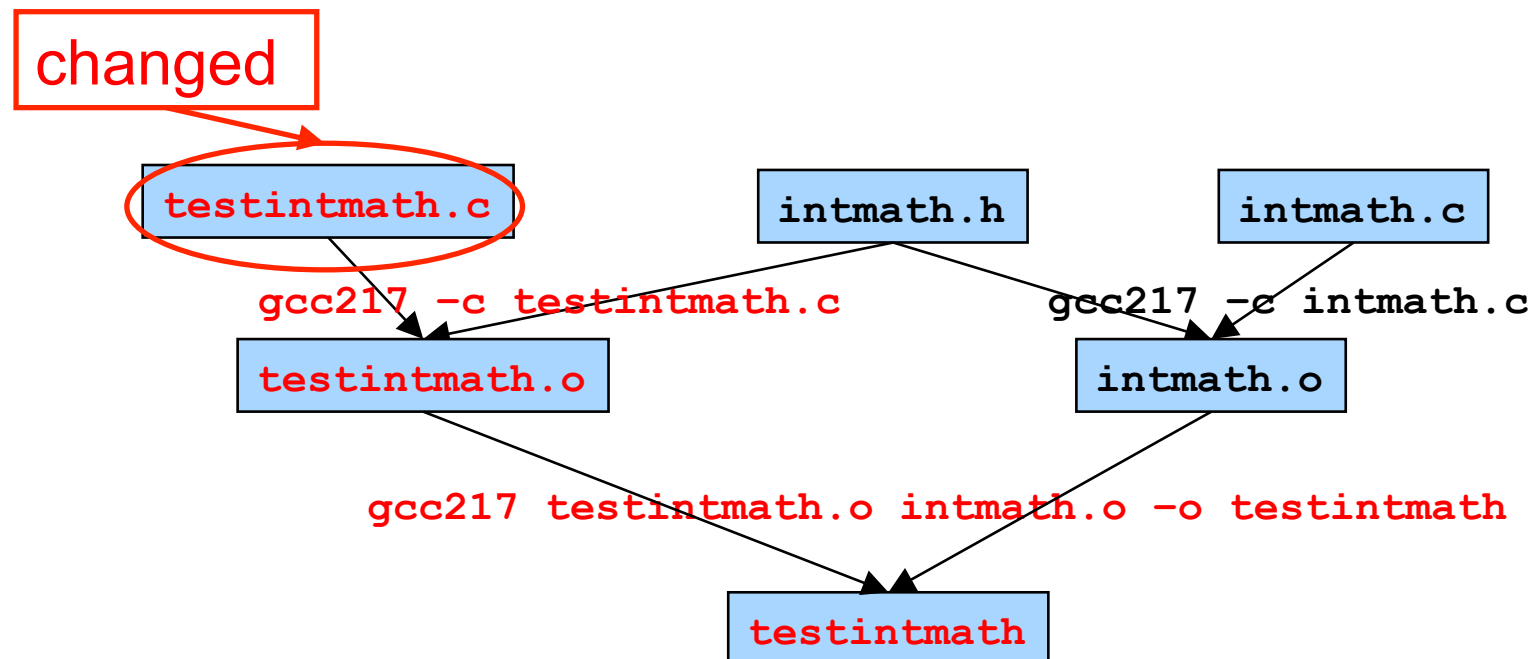




# Partial Builds (cont.)

- 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

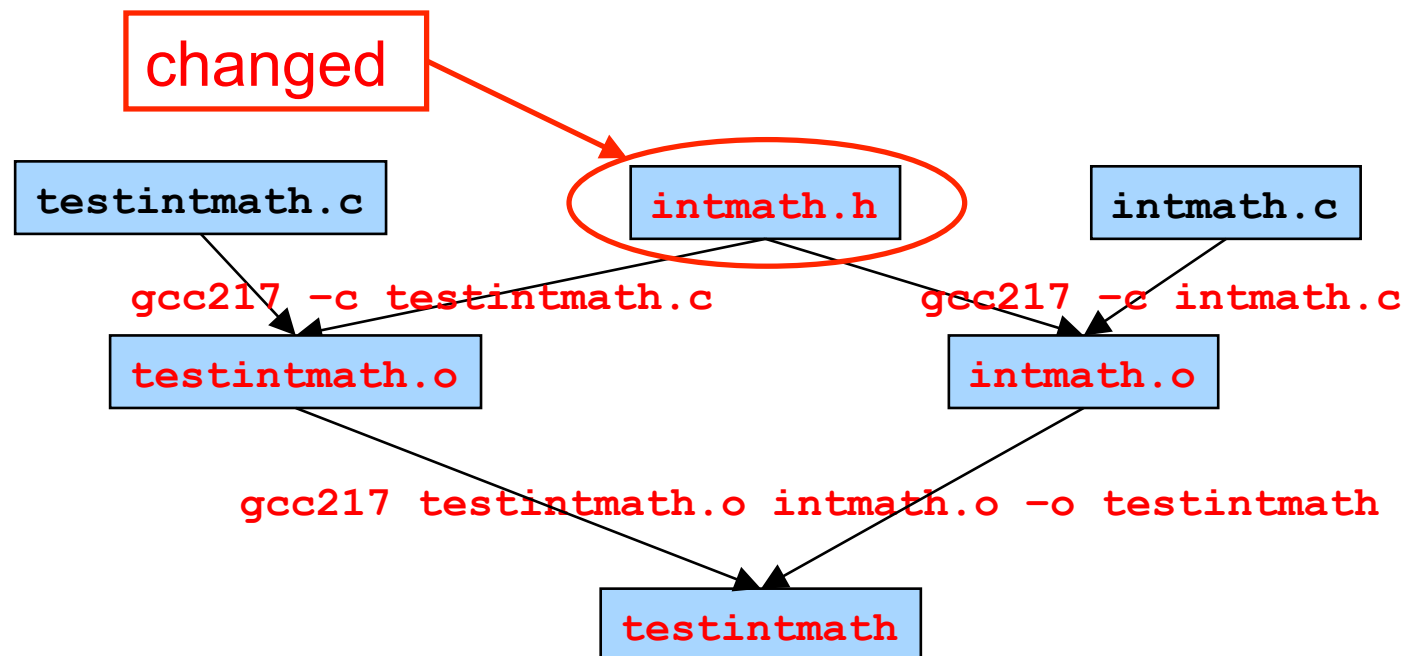




# Partial Builds (cont.)

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`





# 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!**



# Agenda

Motivation for Make

**Make Fundamentals**

Non-File Targets

Macros

Abbreviations (if time)

Pattern Rules (if time)



# The Make Tool

**Who?** Stuart Feldman

**When?** 1976

**Where?** Bell Labs

**Why?** Automate partial builds





# 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*



# 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

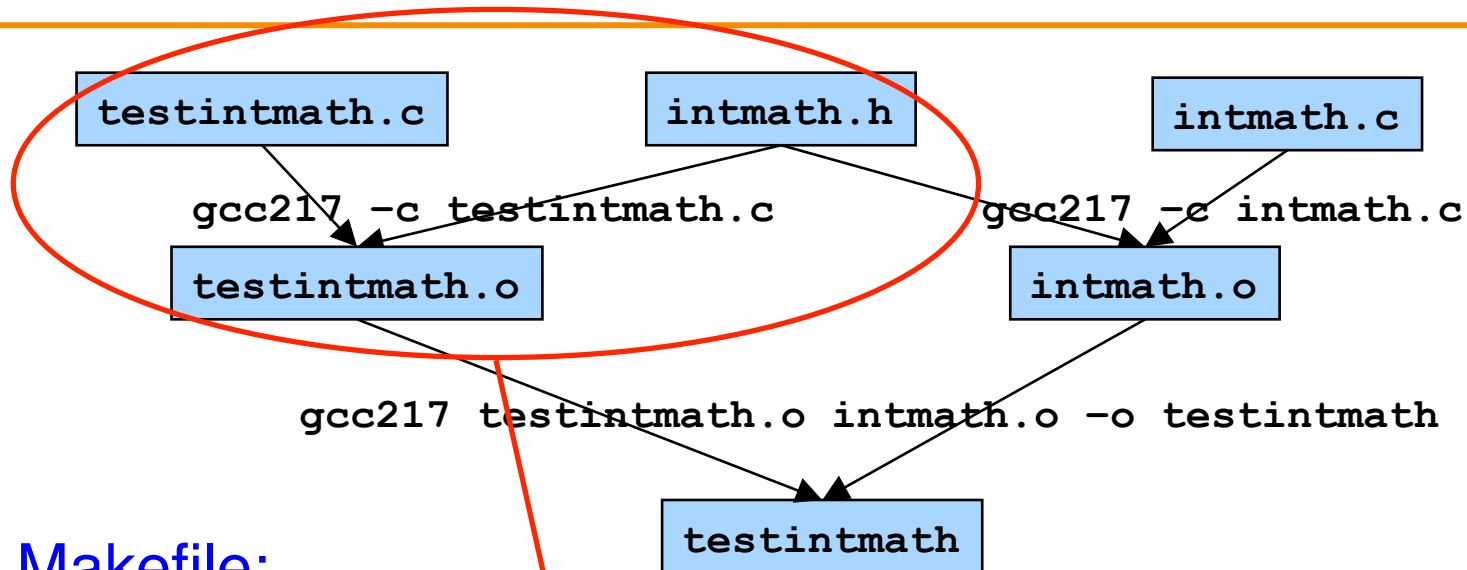
- Build *target* iff it is older than any of its *dependencies*
- Use *command* to do the build

Work recursively; examples illustrate...





# 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
```



# 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



# Agenda

Motivation for Make

Make Fundamentals

**Non-File Targets**

Macros

Abbreviations (if time)

Pattern Rules (if time)



# 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
```



# 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
```



# Version 2 in Action

make observes that “clean” target doesn’t exist; attempts to build it by issuing “rm” command

```
$ 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'.
```

Same idea here, but “clobber” depends upon “clean”

“all” depends upon “testintmath”

“all” is the default target



# Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

**Macros**

Abbreviations (if time)

Pattern Rules (if time)



# 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 -O3
```





# Makefile Version 3

```
# Macros
CC = gcc217
# CC = gcc217m
# CC = gcc217v
CFLAGS =
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O3

# 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
```



# Version 3 in Action

Same as Version 2



# Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

**Abbreviations (if time)**

Pattern Rules (if time)



# 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 $@
```



# Makefile Version 4

```
# Macros
CC = gcc217
# CC = gcc217m
# CC = gcc217v
CFLAGS =
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O3

# 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 $<
```



# Version 4 in Action

Same as Version 2



# Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

Abbreviations (if time)

**Pattern Rules (if time)**



# 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





# 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



# Makefile Version 5

```
# Macros
CC = gcc217
# CC = gcc217m
# CC = gcc217v
CFLAGS =
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O3

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

# 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: intmath.h
intmath.o: intmath.h
```



# Version 5 in Action

Same as Version 2



# Makefile Guidelines

## In a proper Makefile, object file x.o:

- Depends upon x.c
- Does not depend upon any .c file other than x.c
- Does not depend upon any other .o file
- Depends upon any .h file that is #included into x.c
  - Beware of indirect #includes: if x.c #includes a.h, and a.h #includes b.h, then x.c depends upon both a.h and b.h

## In a proper Makefile, an executable binary file:

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



# Making Makefiles

## In this course

- Create Makefiles manually

## Beyond this course

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



# 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



# Make Resources

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

*GNU make*

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



# Summary

## Motivation for Make

- Automation of partial builds

## Make fundamentals (Makefile version 1)

- Dependency rules, targets, dependencies, commands

## Non-file targets (Makefile version 2)

## Macros (Makefile version 3)

## Abbreviations (Makefile version 4)

## Pattern rules (Makefile version 5)