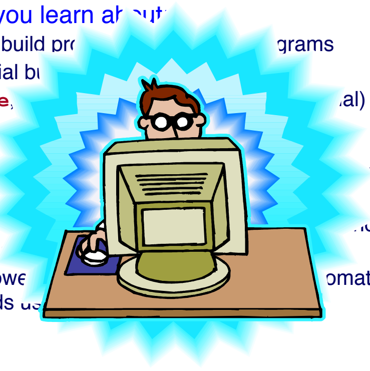
Building

Goals of this Lecture

- Help you learn ab
  - The build pr grams
  - Partial b
  - **make** al) builds
- Why
  - A typically con
  - To ows how to do p
  - A powe omate (partial) builds u



Goals of this Lecture

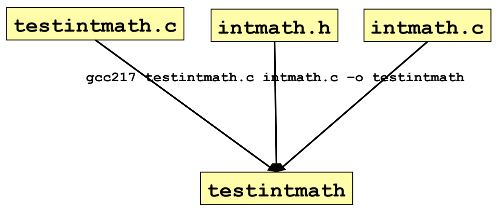
- Help you learn about:
  - The build process for multi-file programs
  - Partial builds of multi-file programs
  - `make`, a popular tool for automating (partial) builds
- Why?
  - A complete build of a large multi-file program typically consumes many hours
  - To save build time, a power programmer knows how to do partial builds
  - A power programmer knows how to automate (partial) builds using `make`

Example of a Three-File Program

- Program divided into three files
  - `intmath.h`: interface, included into `intmath.c` and `testintmath.c`
  - `intmath.c`: implementation of math functions
  - `testintmath.c`: implementation of tests of the math functions
- Recall the program preparation process
  - `testintmath.c` and `intmath.c` are preprocessed, compiled, and assembled separately to produce `testintmath.o` and `intmath.o`
  - Then `testintmath.o` and `intmath.o` are linked together (with object code from libraries) to produce `testintmath`

Motivation for Make (Part 1)

- Building **testintmath**, approach 1:
  - Use one `gcc217` command to preprocess, compile, assemble, and link



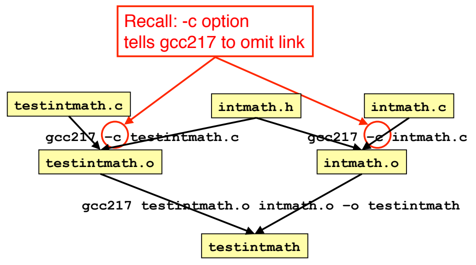
That's not how it's done in the real world…

5

Motivation for Make (Part 2)

- Building **testintmath**, approach 2:
  - Preprocess, compile, assemble to produce .o files
  - Link to produce executable binary file

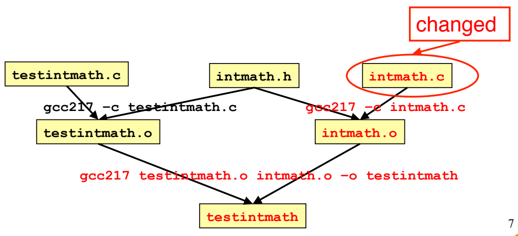
That's how it's done in the real world; Why?...



6

Partial Builds

- Approach 2 allows for partial builds
  - Example: Change `intmath.c`
    - Must rebuild `intmath.o` and `testintmath`
    - Need not rebuild `testintmath.o`

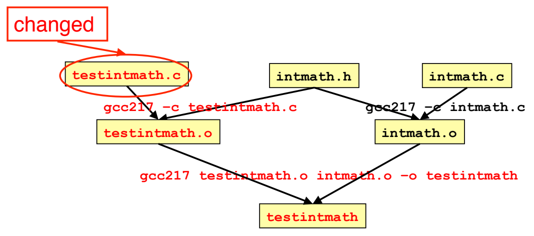


7

Partial Builds (cont.)

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

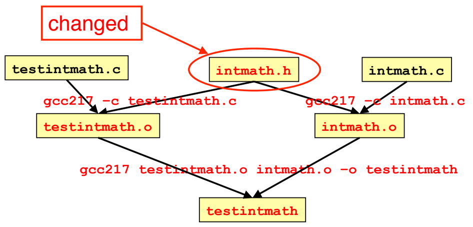
If program contains many .c files, could save many hours of build time



8

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`**



There Oughta be a Tool …

- Observation
  - Doing partial builds manually is tedious and error-prone
  - There oughta be 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`**

Make Fundamentals

- 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`***

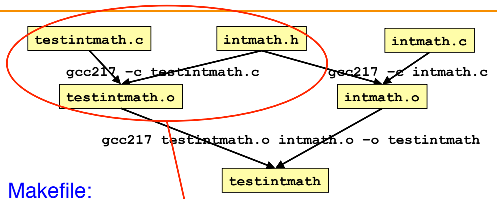
Dependency Rules

- 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
```

Three dependency rules; each captures a fragment of the graph

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

Non-File Targets

- Adding useful shortcuts for the programmer
  - **make all**: create the final binary
  - **make clobber**: delete all temp files, core files, binaries, etc.
  - **make clean**: delete all binaries

```
all: testintmath

clobber: clean
  rm –f *~ \#*\# core

clean:
  rm –f testintmath *.o
```

- Commands in the example
  - **rm –f**: remove files without querying the user
  - Files ending in '**~**' and starting/ending in '**#**' are Emacs backup files
  - **core** is a file produced when a program "dumps core"

Makefile Version 2

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

$ 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

17

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 which build command is used

  `CC = gcc217`
- Example: Make it easy to change build flags

  `CCFLAGS = -DNDEBUG -O3`

18

Makefile Version 3

```
# Macros
CC = gcc217
# CC = gcc217m
CCFLAGS =
# CCFLAGS = -g
# CCFLAGS = -DNDEBUG
# CCFLAGS = -DNDEBUG -O3

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

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

19

Version 3 in Action

- Same as Version 2

20

Sequence We've Seen

1. Initial Makefile with file targets
   testintmath, testintmath.o, intmath.o
2. Non-file targets
   all, clobber, and clean
3. Macros
   CC and CCFLAGS

- See Appendix for 2 additional versions

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

Makefile Gotchas

- Beware:
  - Each command (i.e., second line of each dependency rule) begins with a TAB character, not spaces
  - Use the **`rm -f`** command with caution

Making Makefiles

- In this course
  - Create Makefiles manually
- Beyond this course
  - Can use tools to generate Makefiles automatically from source code
    - See **`mkmf`**, others
  - Can use similar tools to automate Java builds
    - See **`Ant`**

References on Make

- *Programming with GNU Software* (Loukides & Oram) Chapter 7
- *C Programming: A Modern Approach* (King) Section 15.4
- GNU make
  - http://www.gnu.org/software/make/manual/make.html

25

Summary

- Build process for multi-file programs
- Partial builds of multi-file programs
- **`make`**, a popular tool for automating (partial) builds
  - Example Makefile, refined in three steps

26

Appendix: Fancy Stuff

- Some advanced **`make`** features
- Optional in the course…

27

Appendix: Abbreviations

- Abbreviations
  - Target file: $@
  - First item in the dependency list: $<
- Example

```
testintmath: testintmath.o intmath.o
  $(CC) $(CCFLAGS) testintmath.o intmath.o –o testintmath
```

↓

```
testintmath: testintmath.o intmath.o
  $(CC) $(CCFLAGS) $< intmath.o –o $@
```

28

Appendix: Makefile Version 4

```
# Macros
CC = gcc217
# CC = gcc217m
CCFLAGS =
# CCFLAGS = -g
# CCFLAGS = -DNDEBUG
# CCFLAGS = -DNDEBUG -O3

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

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

Appendix: Version 4 in Action

- Same as Version 2

Appendix: Pattern Rules

- Pattern rule
  - Wildcard version of dependency rule
  - Example:

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

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

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

Can omit build command

Appendix: Pattern Rules Bonus

- Bonus with pattern rules
  - First dependency is assumed

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

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

Can omit first dependency

Appendix: Makefile Version 5

```
# Macros
CC = gcc217
# CC = gcc217m
CCFLAGS =
# CCFLAGS = -g
# CCFLAGS = -DNDEBUG
# CCFLAGS = -DNDEBUG -O3

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

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

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

33

Appendix: Version 5 in Action

- Same as Version 2

34

Appendix: Sequence of Makefiles

1. Initial Makefile with file targets
   testintmath, testintmath.o, intmath.o
2. Non-file targets
   all, clobber, and clean
3. Macros
   CC and CCFLAGS
4. Abbreviations
   $@ and $<
5. Pattern rules
   %.o: %.c

35