COS 217: Introduction to Programming Systems

Managing Modularity

Building Multifile Programs with `make`

PRINCETON UNIVERSITY

Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

Multi-File Programs

intmath.h (interface)

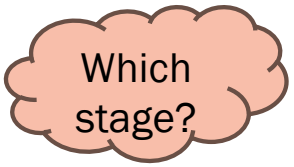
```
#ifndef INTMATH_INCLUDED
#define INTMATH_INCLUDED
int gcd(int i, int j);
int lcm(int i, int j);
#endif
```

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;
}
```

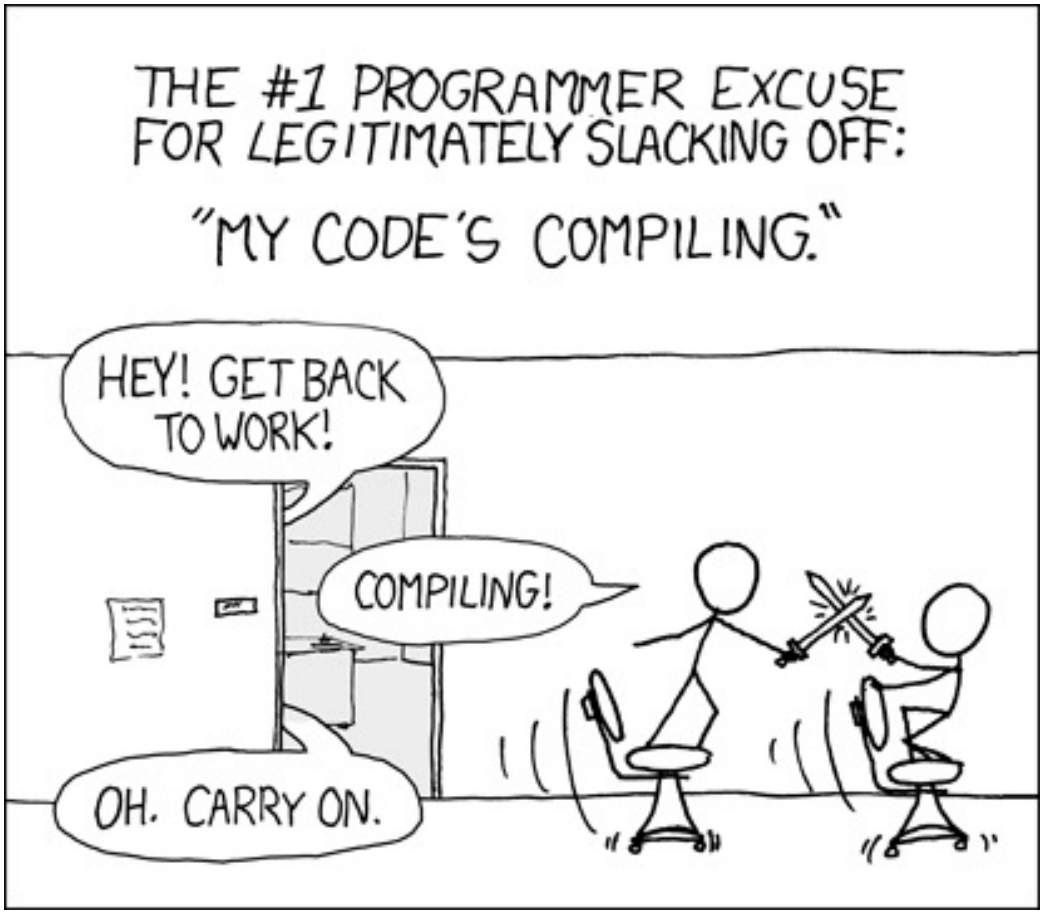
testintmath.c (client)

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

int main(void)
{
   int i, 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

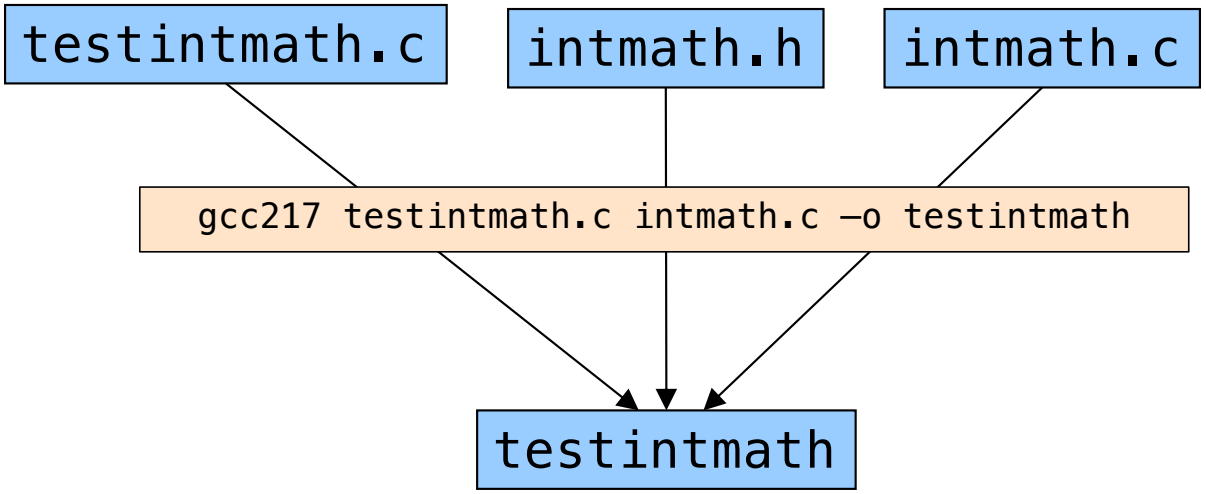
Motivation for `make`



https://xkcd.com/303/

Building `testintmath`: Approach 1 ("shortcut version")

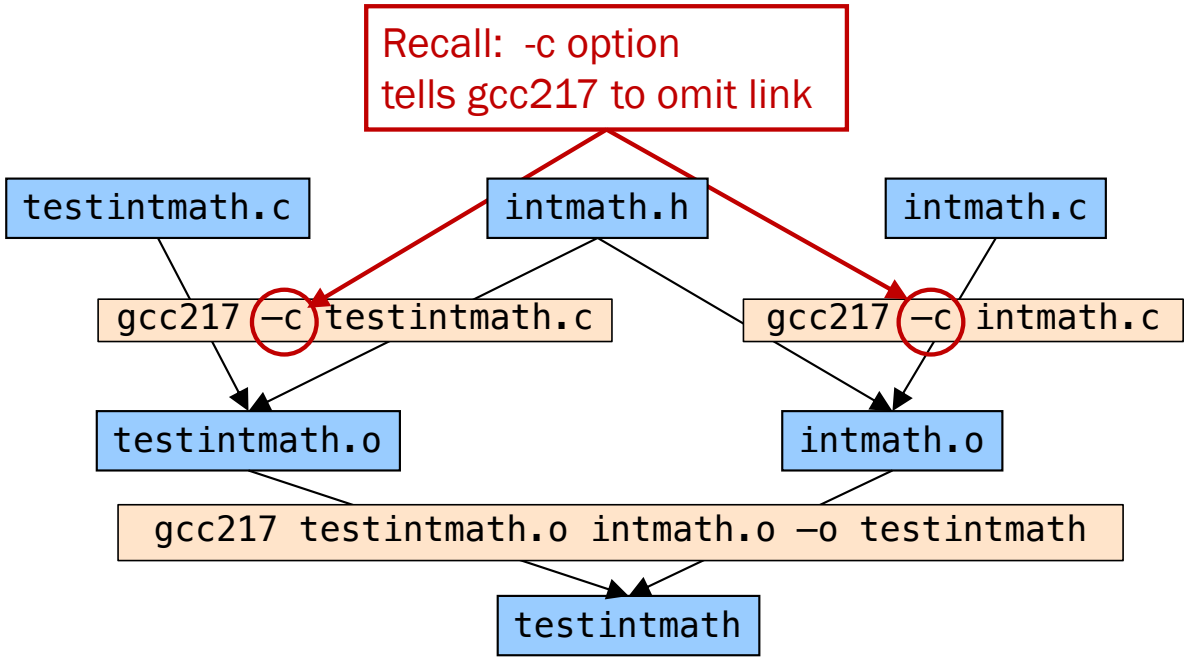
One `gcc217` command to preprocess, compile, assemble, and link



Building `testintmath`: Approach 2

Preprocess, compile, assemble to produce .o files

Link to produce executable binary file



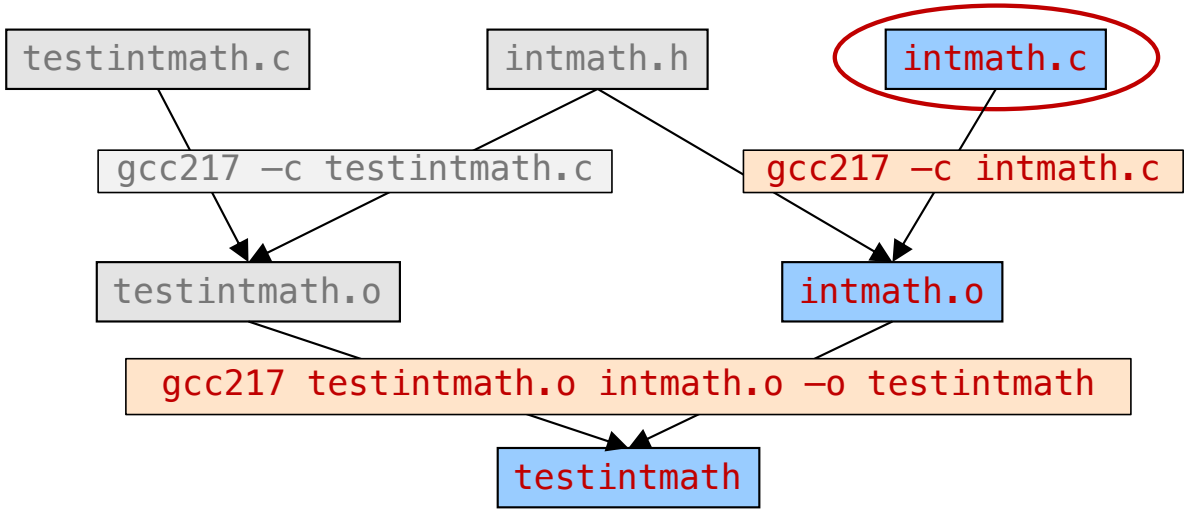
Allows us to preserve .o files across builds, and not have to rebuild them if nothing they depend on has changed

Partial Builds

Approach 2 allows for **partial builds**

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

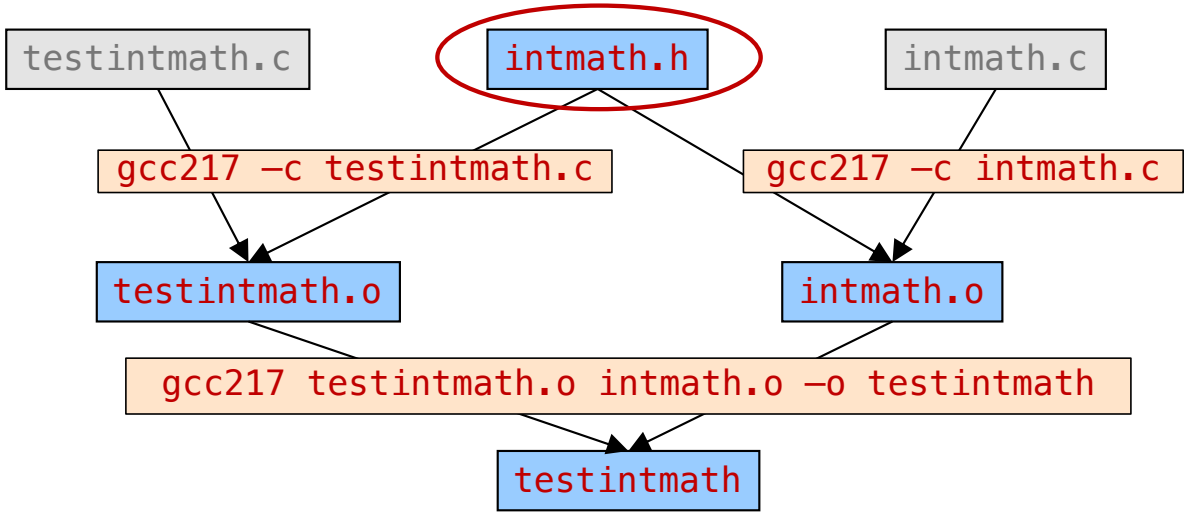
If program contains many files, could save hours of build/test time



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`



Wouldn't It Be Nice If We had a Tool to Automate This

Observation

- Doing partial builds manually is tedious and error-prone

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
- What it does: Figures out which components do and don't need to be rebuilt
- 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`!

Obligatory Princeton Context

Stuart Feldman ’68 (Astro)

- Chief Scientist at Schmidt Futures
- Former President of ACM
- AAAS, IEEE, and ACM fellow
- Board Chair of CMDiT
  Fostering Innovation Through Inclusion

Created `make` at Bell Labs in 1976

Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

Make Command Syntax

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

- `makefile`
  - Textual representation of dependency graph
  - Contains **dependency rules**, that together make up the dependency graph
  - Default name is `makefile`, then `Makefile`

- `target`
  - What `make` should build
  - Usually: .o file(s) or executable binary file. Makefile specifies targets that can be built
  - If not specified, default is to build first target defined in `makefile`

Dependency Rules in Makefile

Dependency rule syntax

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

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

Dependency rule semantics

- Build `target` if it doesn't exist
- Rebuild `target` iff it is older than at least one of its `dependencies`
- Use `command` to do the build
- Work recursively; examples illustrate…

Make gotcha: tab means tab **not** *k* spaces

```
<tab>command
```

The first character of the line with the command must be an *actual tab character*, ASCII character 9. Cryptic error for failing to do so:

```
*** missing separator.  Stop.
```

Feldman explains the genesis:

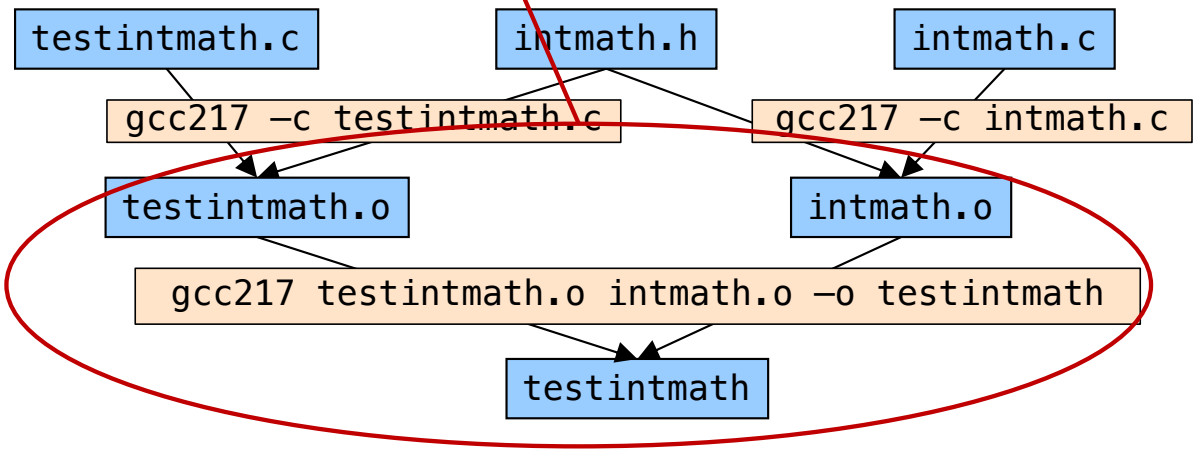
"Within a few weeks of writing Make, I already had a **dozen** friends who were using it" ... "I didn't want to upset them" ... "So instead I wrought havoc on tens of millions."

--Cobbled from Brian Kernighan's *UNIX: A History and a Memoir* and Michael Stillwell

Makefile Version 1

Makefile

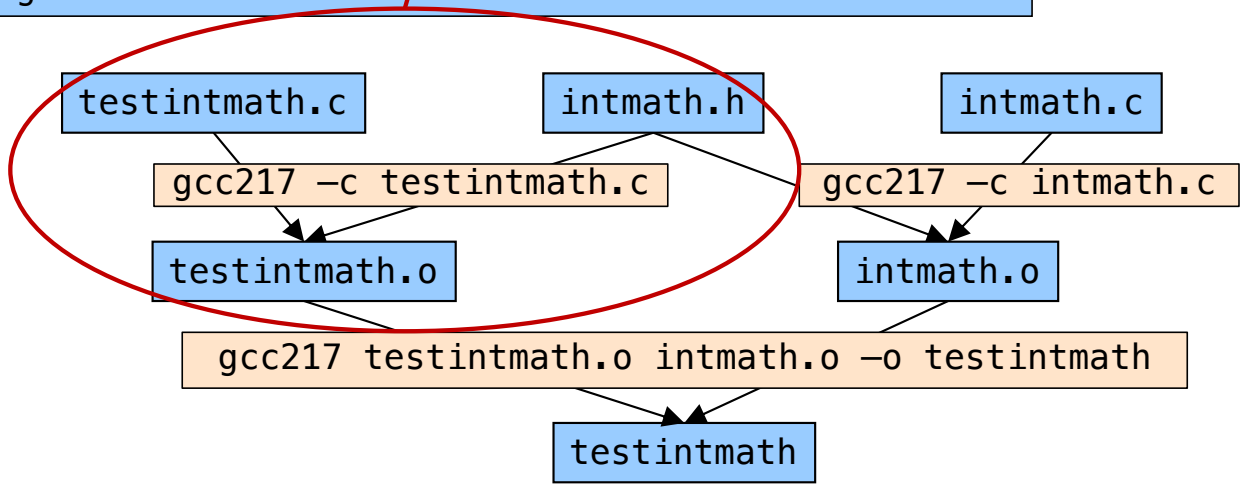
```
testintmath: testintmath.o intmath.o
  gcc217 testintmath.o intmath.o -o testintmath

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

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



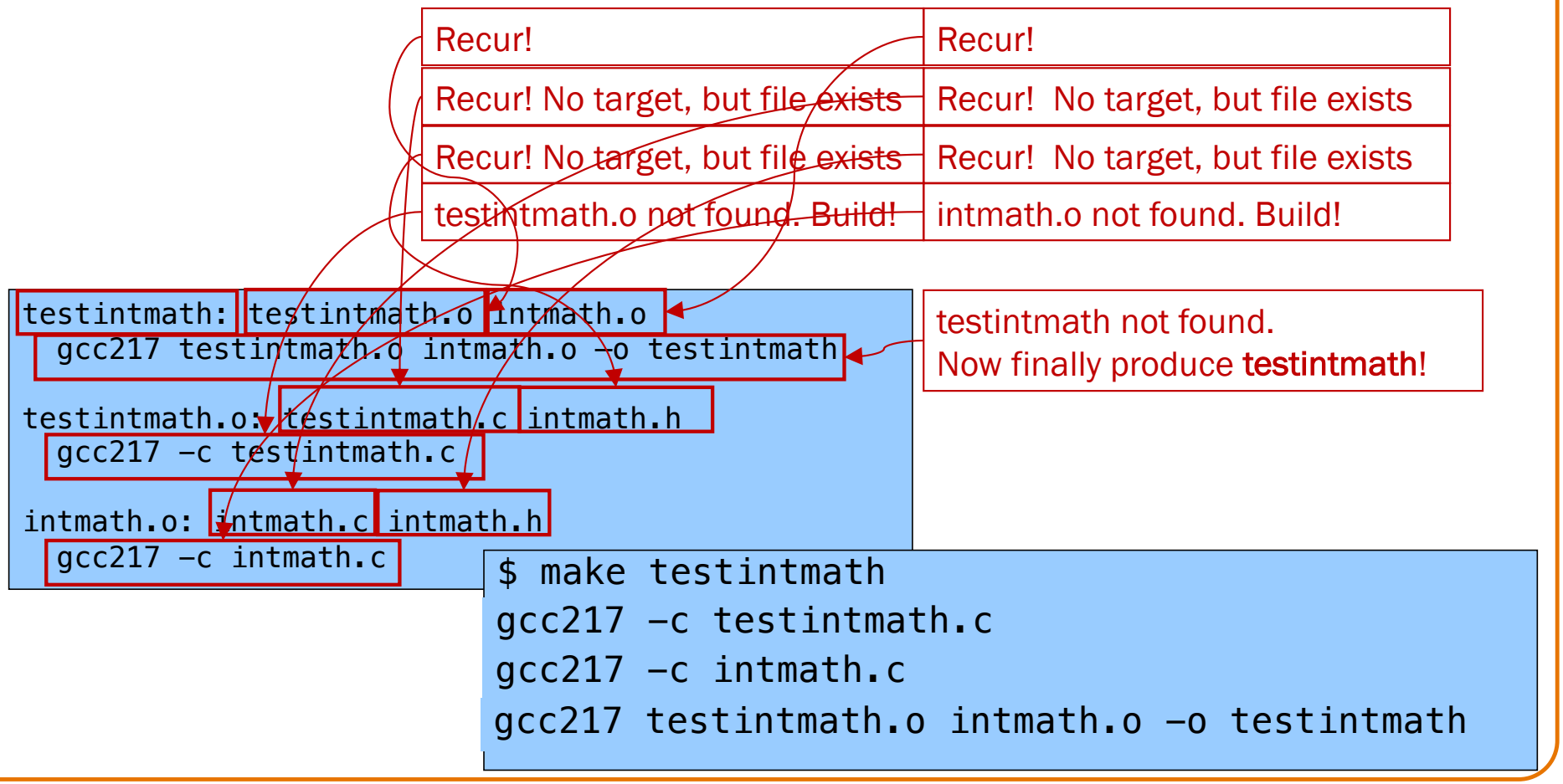
Makefile Version 1

Makefile

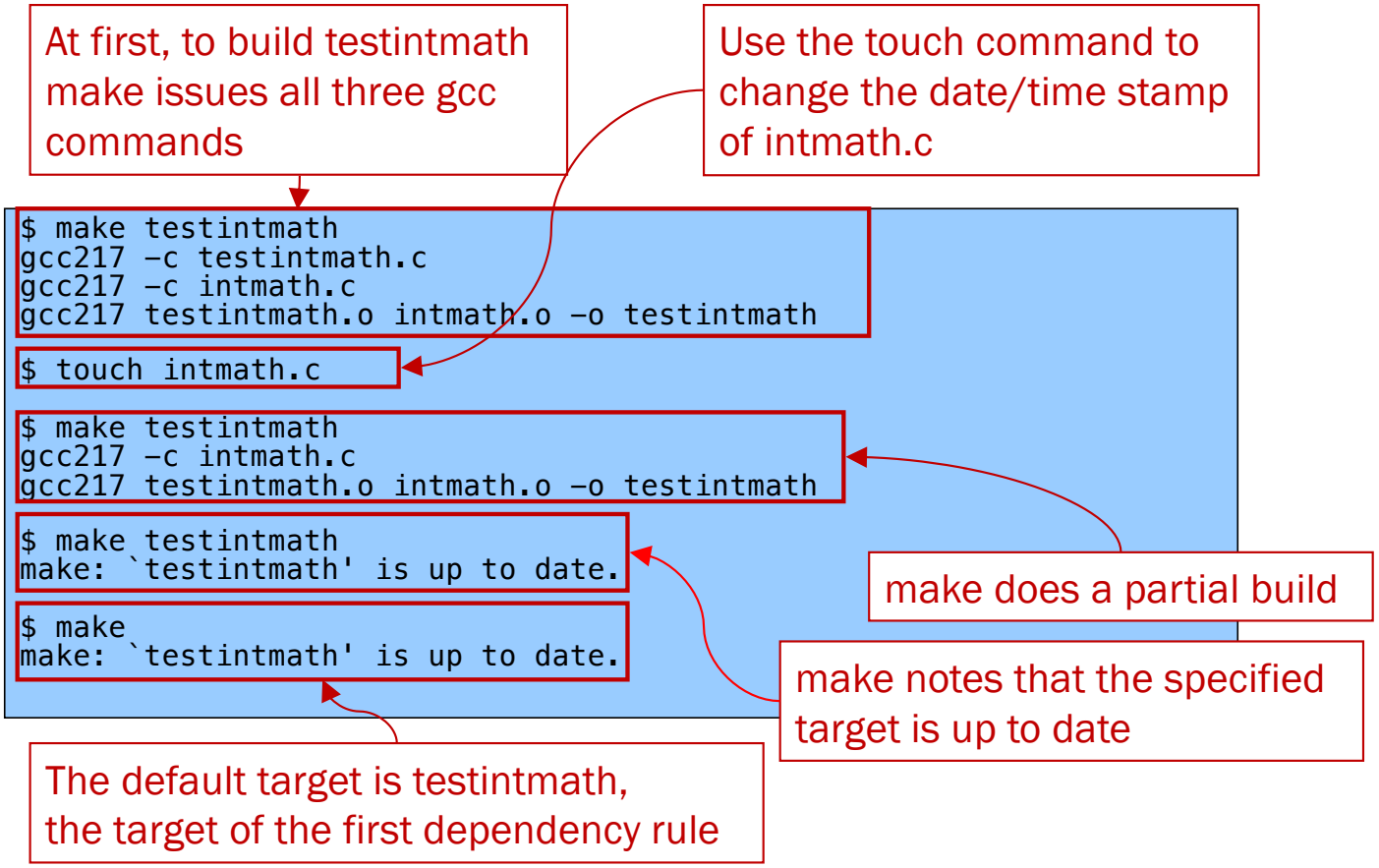
```
testintmath: testintmath.o intmath.o
  gcc217 testintmath.o intmath.o -o testintmath

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

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



Version 1 in Action

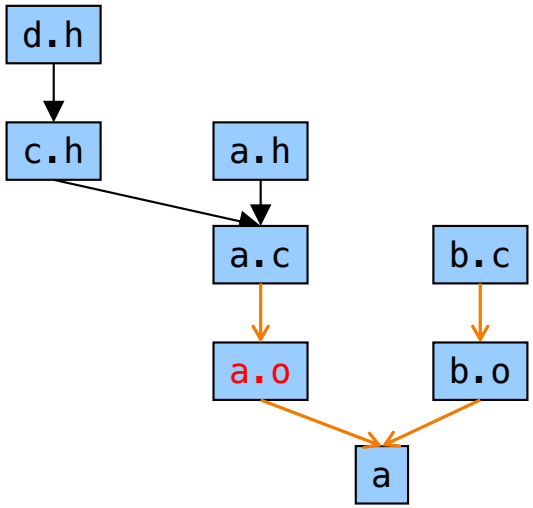


Version 1 in Action



make up your mind

Q: If you were making a `Makefile` for this program (where black solid arrows are #include relationships and orange unfilled arrows are build process relationships), what should `a.o` depend on?



A. a

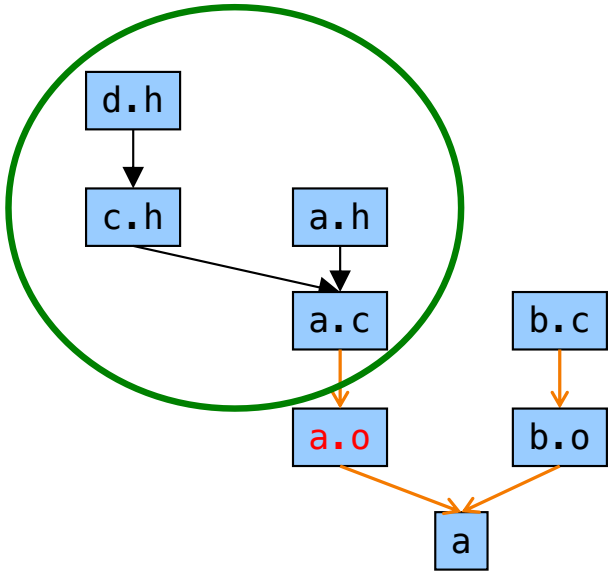
B. a.c

C. a.c b.c

D. a.h c.h d.h

E. a.c a.h c.h d.h

Makefile Guidelines



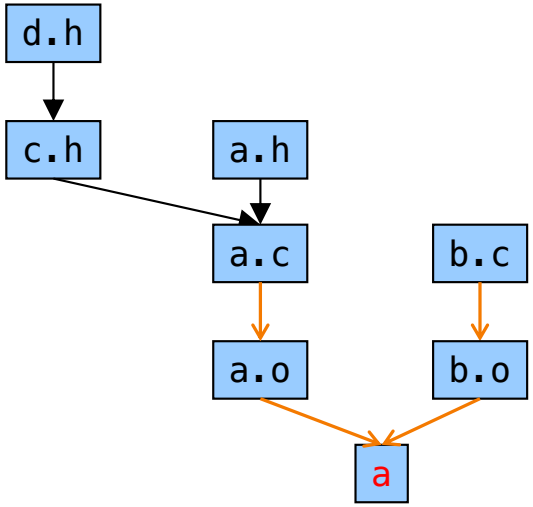

```
a.o: a.c a.h c.h d.h
```

In a properly constructed 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**

building understanding

Q: If you were making a `Makefile` for this program (where black solid arrows are #include relationships and orange unfilled arrows are build process relationships), what should the executable `a` depend on?



A. a.o b.o

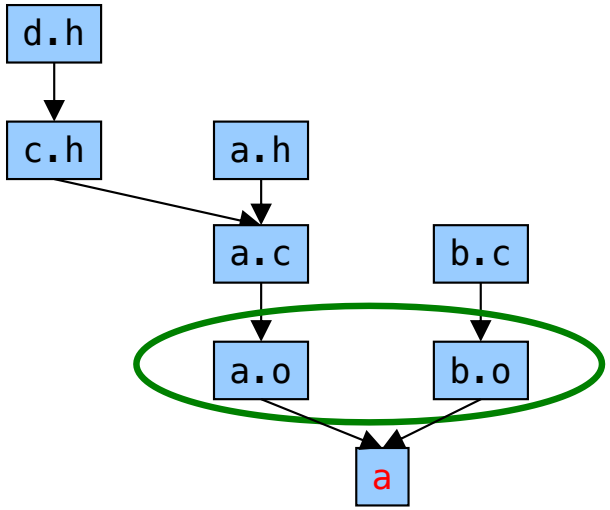
B. a.o b.o a.c b.c

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

Makefile Guidelines




```
a: a.o b.o
```

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

Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

Non-File Targets (aka “pseudotargets”)

Take advantage that make doesn't check that a target actually gets built to add useful shortcuts

Commonly defined non-file targets (but “all”, “clean”, ”clobber” are not syntactically special):

- `make all`: create the final executable binary file(s), often the first target listed in the Makefile, perhaps with multiple executables that the Makefile builds
- `make clean`: delete all .o files, executable binary file(s)
- `make clobber`: delete all .o files, executable(s), and assorted development cruft (e.g., Emacs backup files)

```
all: testintmath

clobber: clean
  rm -f *~ \#*\#

clean:
  rm -f testintmath *.o
```

- `rm -f`: remove files without querying user. Files ending in ‘~’ and starting/ending in ‘#’ are Emacs backup and autosave files

Typical first target in makefile is all or help (provides info but doesn’t build anything)

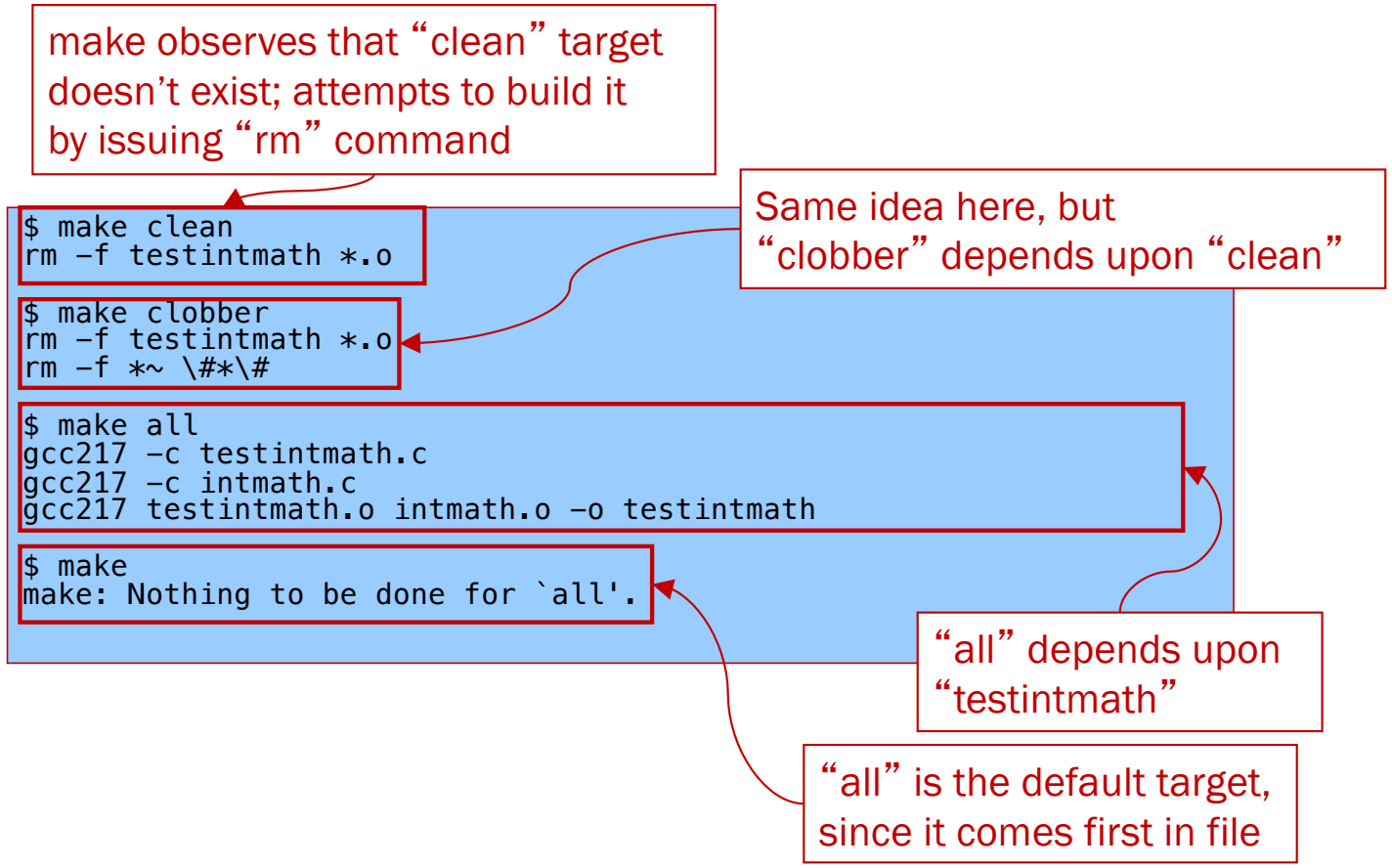
clean, clobber, all are treated like files (but don’t exist), but make doesn’t that care they don’t get built

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
```

- We don't use make clean too often because we don't want to get rid of .o files (we like partial builds)
- We could also have a rule for removing the *~ and \#*\# files without first removing .o files

Version 2 in Action



Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

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
YACC = bison -d -y
#YACC = yacc -d
```

Example: Make it easy to change build flags

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

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
```

Version 3 in Action

Same as Version 2

More Makefile Gotchas

Beware:

- Bears repeating: each command (second line of each dependency rule) must begin with a tab character, not spaces – configure your editor accordingly!
- 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
  (Users are likely to just type `make` , out of habit or ignorance.)

Making Makefiles

In this course

- Create Makefiles manually
- Perhaps start from the Makefiles from this lecture?

Beyond this course

- Can use tools to generate Makefiles
  - See `mkmf`, among others
- Copy, paste, edit existing Makefiles

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.
We'll never ask you to write these.

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 almost certainly confusing).
Not allowed in this class

If One Did .... Makefile Version 4

```
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
```

```
# 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
```

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

```
testintmath: testintmath.o intmath.o
```

Progressively terser but more confusing. Just don't.

Implicit Rule Gotcha

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
```


Again, implicit rules not allowed in this class. Be explicit

Make Resources

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

C Programming: A Modern Approach (King) Section 15.4