Makefile

Goals

- 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 (*e.g.* Linux source code)
  - To save build time, a good programmer knows how to do partial builds
  - A good programmer knows how to automate (partial) builds using **make**

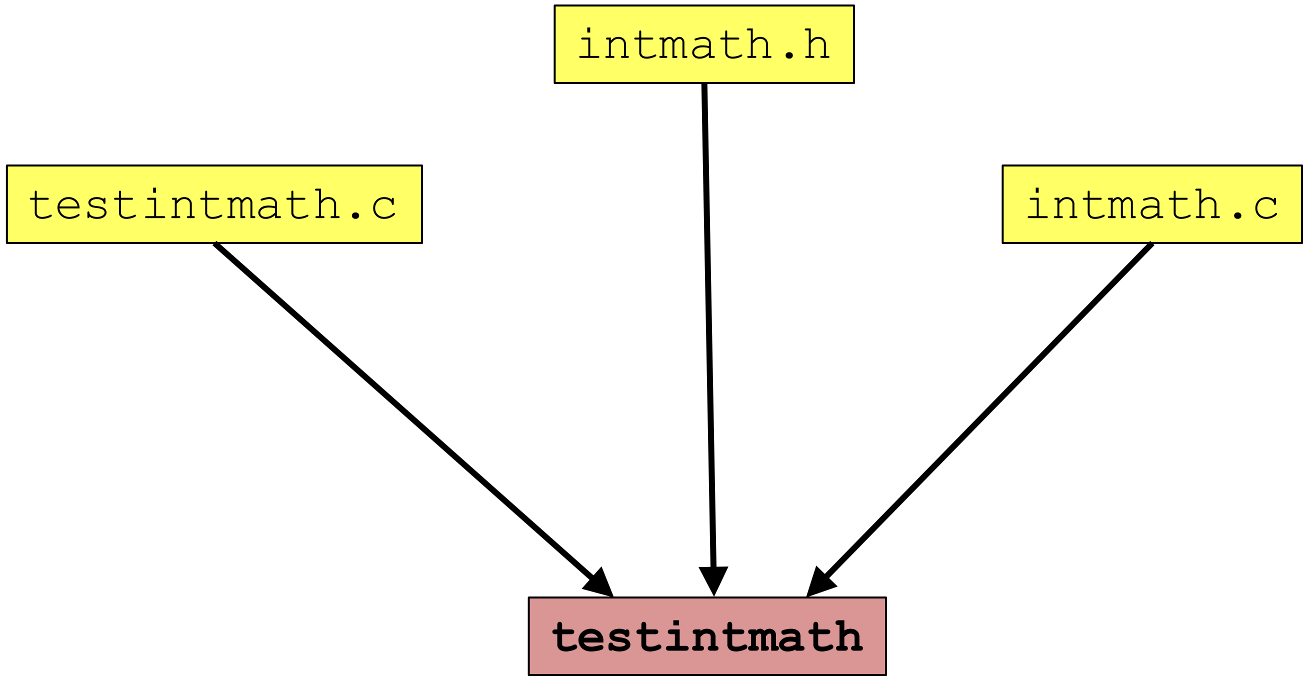
Example: intmath lib

- Program divided into 3 files
  - `intmath.h`
  - `intmath.c`
  - `testintmath.c`

- Recall the program prep process
  - `testintmath.c` & `intmath.c` are preprocessed, compiled and assembled separately → `testintmath.o` & `intmath.o`
  - Then `testintmath.o` & `intmath.o` are linked together (with object code from libs) to produce `testintmath`

make Motivation I

- Building `testintmath`, Approach 1:

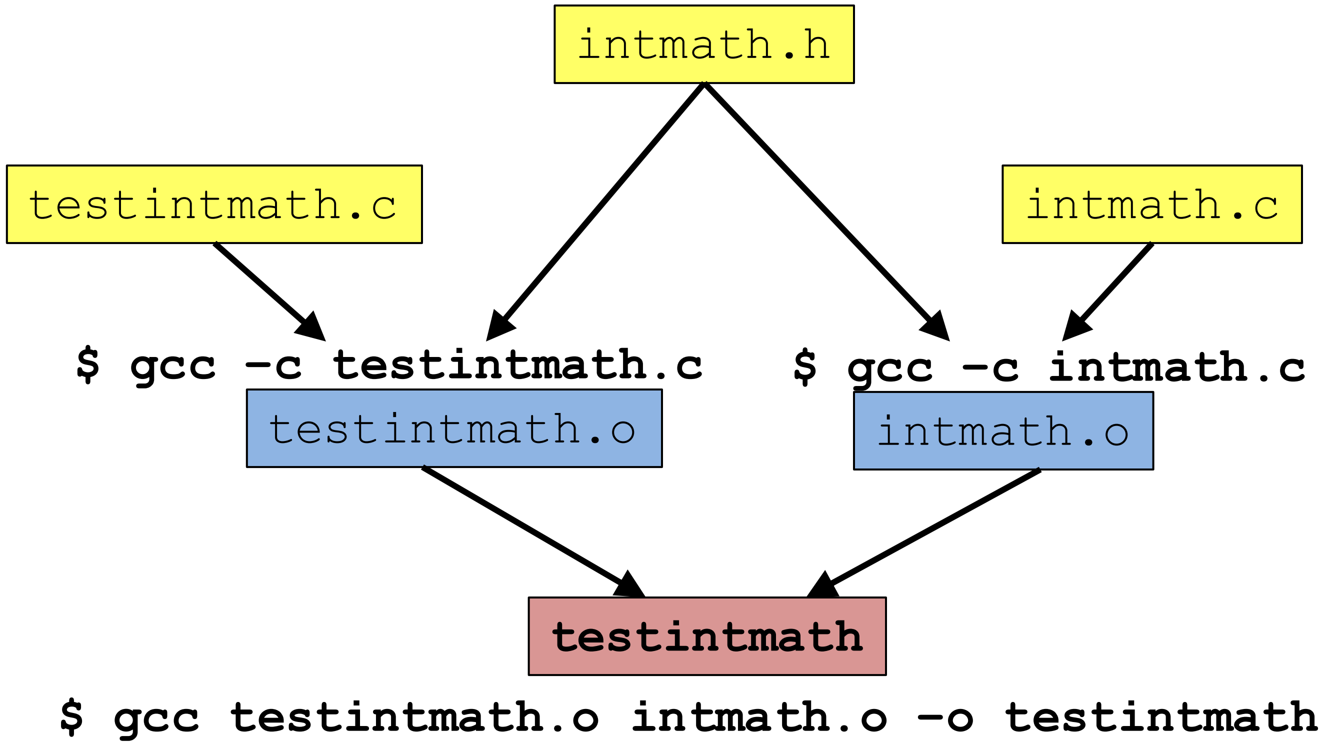


`$ gcc testintmath.c intmath.c -o testintmath`

VERY INCONVENIENT AS YOU WILL SEE SOON

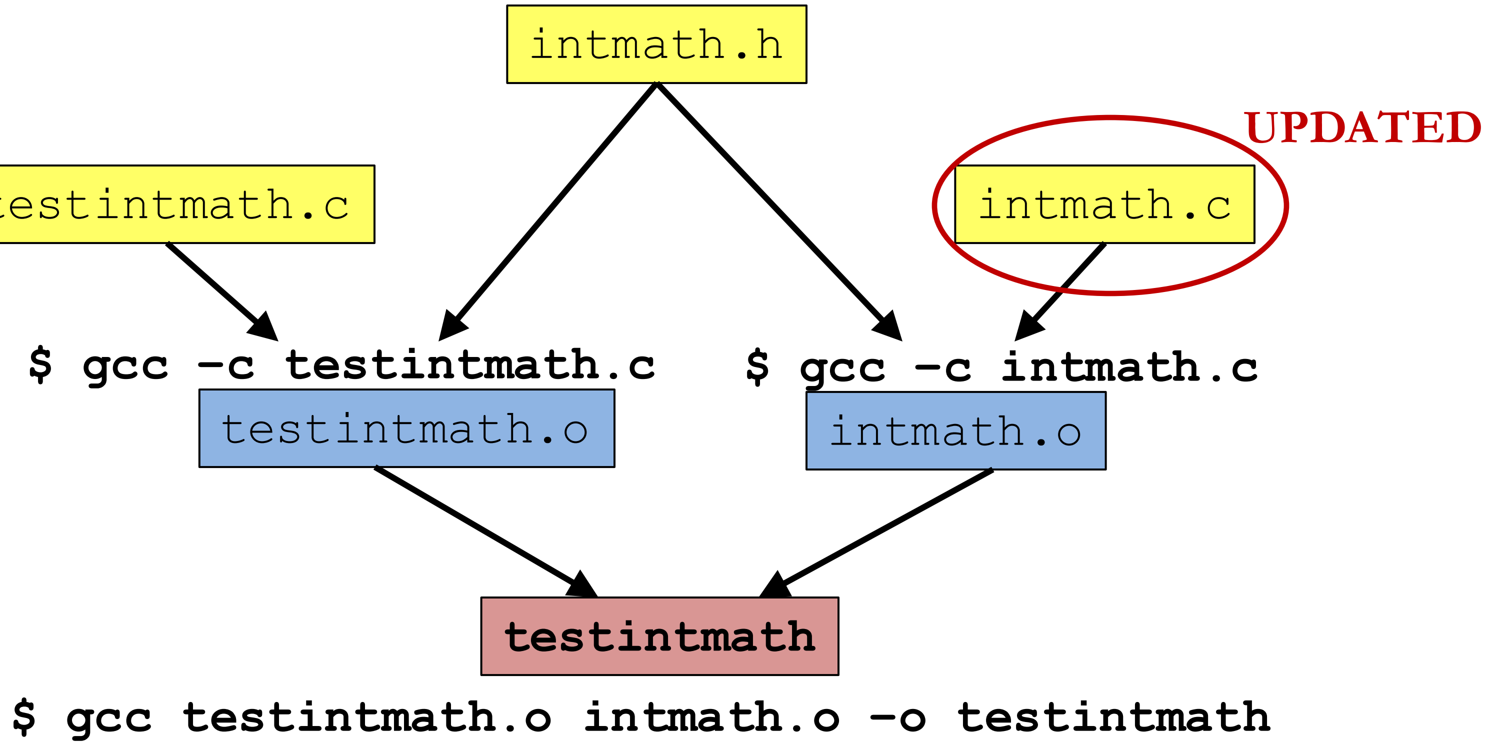
make Motivation II

- Approach 2:



Partial Builds

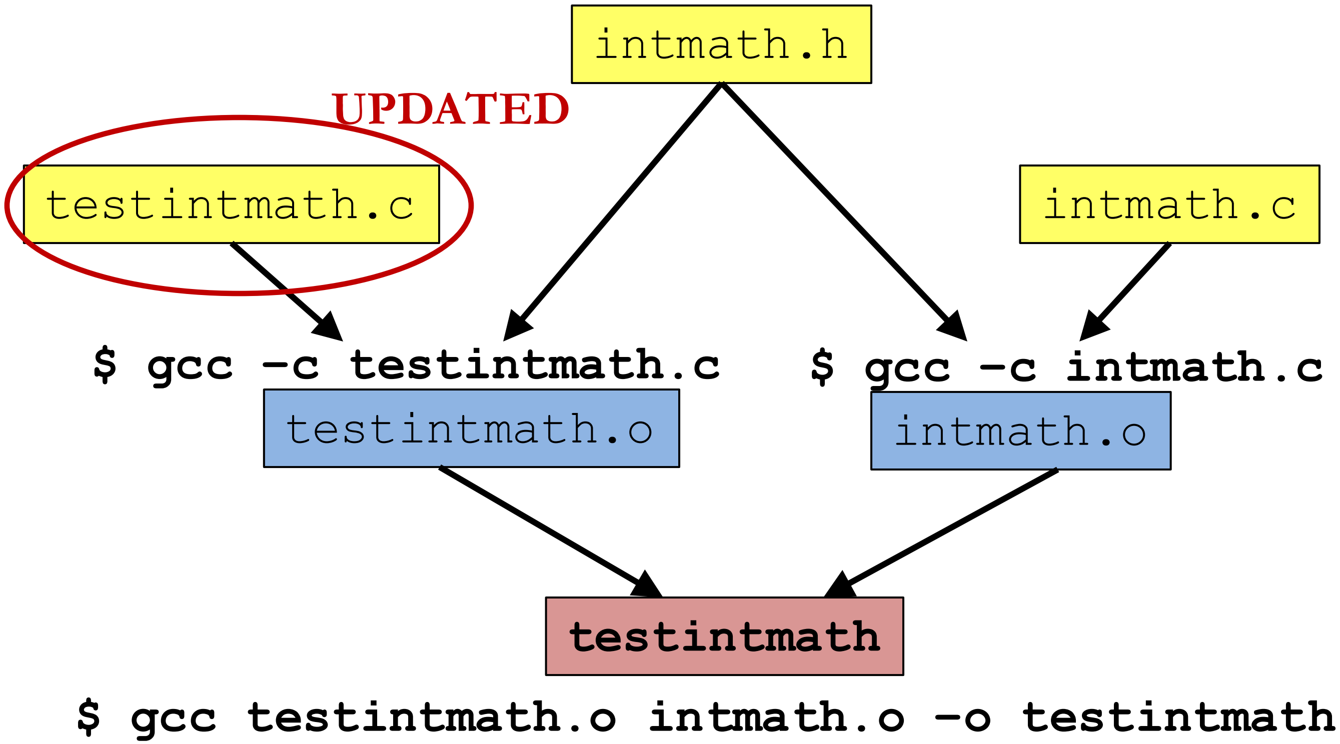
- Partial builds now possible:



testintmath.o DOES NOT NEED TO BE REBUILT

Partial Builds

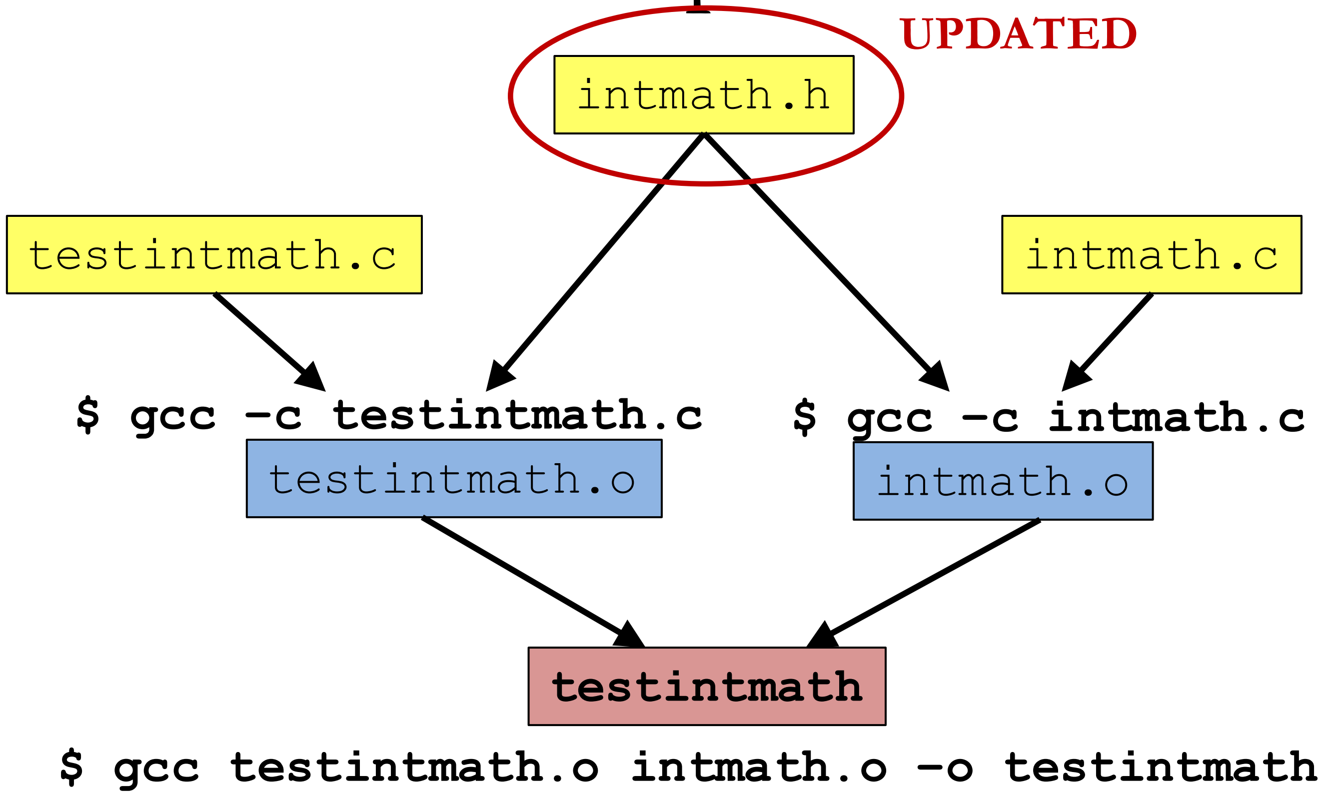
- Partial builds now possible:



MANY HOURS OF BUILD TIMES SAVED!

Partial Builds

- Partial builds now possible:



HOWEVER CHANGING `intmath.h` IS MORE DRAMATIC

Observation

- Doing partial builds manually is tedious & error-prone
- Make tool:
  - Input:
    - Dependency graph (like previously shown)
      - Specifies file dependencies
      - Specifies commands to build each file from its dependents
    - File timestamps
  - Algorithm:
    - **If file B depends on A & timestamp of A is newer than timestamp of B, then rebuild B using the specified command**

Make Fundamentals

- Command

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

- makefile
  - Textual representation of dependency graph
  - Contains dependency rules
  - Default name is Makefile
- Target
  - What **`make`** should build
  - Usually it's the .o file or an binary file
  - Default is fist one defined in the makefile

Dependency Rules

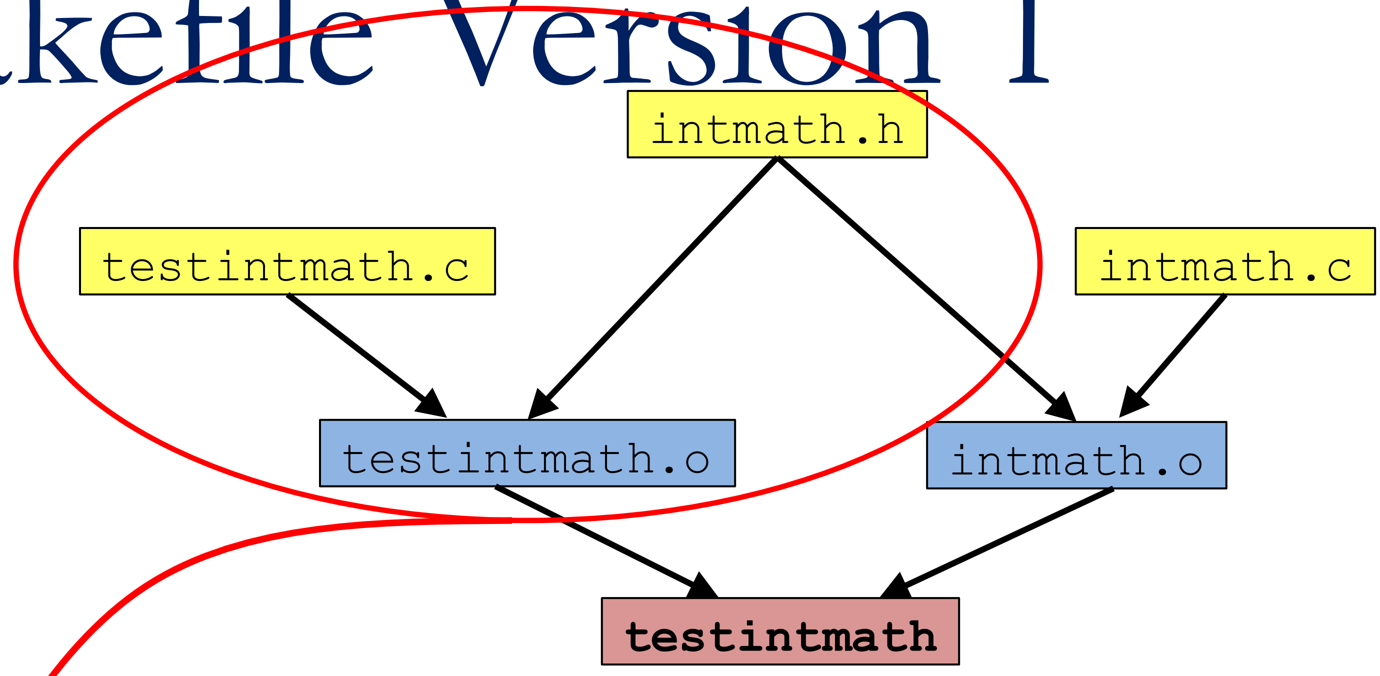
- Synatax

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

  - **target**: the file you want to build
  - **dependencies**: the list of files the target depends on
  - **command:** what to execute to create the target
- Semantics
  - Build target iff it is older than any of its **dependencies**
  - Use **command** to build

- Work recursively

Makefile Version 1




```
testintmath: testintmath.o intmath.o
   gcc testintmath.o intmath.o –o testintmath

testintmath.o: testintmath.c intmath.h
   gcc –c testintmath.c

intmath.o: intmath.c intmath.h
   gcc –c intmath.c
```

Version 1 in Action

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

$ touch intmath.c

$ make testintmath
gcc -c intmath.c
gcc testintmath.o intmath.o -o testintmath

$ make testintmath
make: `testintmath' is up to date

$ make
make: `testintmath' is up to date
```

ISSUES ALL 3 GCC COMMANDS USING THE DEPENDENCY GRAPH

UPDATE INTMATH.C TIMESTAMP

PARTIAL BUILD

THE DEFAULT TARGET IS TESTINTMATH, THE TARGET OF THE FIRST DEPENDENCY RULE

Non-File Targets

- Useful shortcuts
  - **make all**: create the final binary
  - **make clobber**: delete all temp, core and binary files
  - **make clean**: delete all binary files

- Commands in the example
  - **rm -f**: remove files without querying the user. Files ending in '~' and '#' (emacs baclup files)
  - **core** file is generated when a program 'dumps core'

```
all: testintmath

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

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

Makefile Version 2

```
# Dependency fules 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
   gcc testintmath.o intmath.o -o testintmath

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

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

Version 2 in Action

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

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

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

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

"CLOBBER" DEPENDS ON "CLEAN".

ALL DEPENDS ON TESTINTMATH

ALL IS THE DEFAULT TARGET

Macros

- Similar to C preprocessor's #define

- Example:

```
CC=gcc
CCFLAGS= -DNDEBUG -O3
```

Makefile Version 3

```
CC=gcc
#CC=gcc209
CCFLAGS=
#CCFLAGS=-g
#CCFLAGS=-DNDEBUG -g

# Dependency fules 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
```

Version 3 in Action

- Same as Version 2

Abbreviations

- Target file: `$@`
- First item in the dependency list: `$<`
- All items in the dependency list: `$?`

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

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

Makefile Version 4

```
CC=gcc
#CC=gcc209
CCFLAGS=
#CCFLAGS=-g
#CCFLAGS=-DNDEBUG -g

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

Version 4 in Action

- Same as Version 2

Pattern Rules

- Wildcards

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

  - To build .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) $? -o $@
testintmath.o: testintmath.c intmath.h
intmath.o: intmath.c intmath.h
```

Pattern Rules Bonus

- First dependency is assumed

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

FIRST DEPENDENCY IS SKIPPED

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

Makefile Version 5

```
CC=gcc
#CC=gcc209
CCFLAGS=
#CCFLAGS=-g
#CCFLAGS=-DNDEBUG -g

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

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

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

Version 5 in Action

- Same as Version 2

Makefile Guidelines

- In a Makefile, any object file x.o
  - depends on x.c
  - does not depend on any .c file other than x.c
  - does not depend on any .o file
  - depends on any .h file that is #included in x.c
- In a Makefile, any binary
  - depends on the .o files
  - does not depend <u>directly</u> on any .c files
  - does not depend <u>directly</u> on any .h files

Makefile 'Gotchas'

- Each command (*i.e.*, second line of each dependency rule) begins with a <TAB> character

- Use the '`rm -f`' command with caution

Auto-generating Makefiles

- You can use tools to generate Makefiles automatically from source code
  - See **mkmf** (C)
  - See **Ant** (Java)

- We will not cover these in this course

Makefile References

- 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

Summary

- Initial Makefile with file targets

  testintmath, testintmath.o, intmath.o

- Non-file targets

  all, clobber and clean

- Macros

  CC and CCFLAGS

- Abbreviations

  $@, $? and $<

- Pattern rules

  %.o: %.c

Performance

Goals

- How to improve memory & CPU bandwidth
  - GPROF execution profiler
- Why?
  - Usually a small fragment of the code consumes major chunk of CPU time and/or memory
  - A good program knows how to identify and improve such fragments

Questions

- How slow is my program?
- Where is my program slow?
- Why is my program slow?
- How can I make my program run faster?
- How can I make my program use less memory?

However…

- Code may become less
  - clear
  - maintainable
- May confuse debuggers
- May inject bugs

When to improve

"The first principle of optimization is

don't!

Is the program good enough already? Knowing how a program will be used and the environment it runs in, is there any benefit to making it faster?"

--Kernighan & Pike

The 5 techniques

- Let's consider them one at a time…

Timing Studies (1)

- Use the Unix `time` command

```
$ time sort < bigfile.txt > output.txt
real    0m12.977s
user    0m12.860s
sys     0m0.010s
```

  - Real: Wall-clock time b/w program invocation & termination
  - User: CPU time spent executing the program
  - System: CPU time spent within the OS on the program's behalf

Timing Studies (1)

- To time parts of program, use gettimeofday() function (time since Jan 1, 1970)
- Not defined in C90 standard

```
#include <sys/time.h>

struct timeval startTime;
struct timeval endTime;
double wallClockSecondsConsumed;

gettimeofday(&startTime, NULL);
/* execute some code here */
gettimeofday(&endTime, NULL);
wallClockSecondsConsumed =
              endTime.tv_sec - startTime.tv_sec +
              1.0E-6*(endTime.tv_usec - startTime.tv_usec);
```

Timing Studies (1)

- To time parts of program, call a function to compute **CPU time** consumed
- *e.g* **`clock()`** function – defined by C90 standard

```
#include <time.h>

clock_t startClock;
clock_t endClock;
double cpuSecondsConsumed;

startClock = clock();
/* execute some code here */
endClock = clock();
cpuSecondsConsumed =
       ((double)(endClock - startClock))/CLOCKS_PER_SEC;
```

Identify Hot Spots

- Gather statistics about your program's execution
  - how much time did a function take for execution?
  - how many times was a specific function called?
  - how many times was a specific line executed?

- Execution profiler:
  - **gprof** (GNU Performance Profiler)

GPROF (2)

- Step 1: Instrument the program

```
$ gcc209 -pg testsymtable.c symtablelist.c -o testsymtable
```

  – Adds profiling code to testsymtable

- Step 2: Run the program

```
$ ./testsymtable 10000
```

  – Creates file gmon.out containing statistics

- Step 3: Run the program

```
$ gprof ./testsymtable > report_list
```

  – Uses testsymtable and gmon.out to create textual report

- Step 4: Examine the report

```
$ less report_list
```

GPROF (2)

- symtablelist.c

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.00	0.48	0.48	32073	0.01	0.01	SymTable_eexists
0.00	0.48	0.00	63108	0.00	0.00	assure
0.00	0.48	0.00	11020	0.00	0.01	SymTable_get
0.00	0.48	0.00	11018	0.00	0.01	SymTable_put
0.00	0.48	0.00	10010	0.00	0.01	SymTable_remove

- name: name of function
- %time: %age of time spent executing this function
- cum. secs: [not relevant]
- self seconds: time spent executing this function
- calls: number of times function was called (excluding recursion)
- self ms/call: avg time per execution (excluding descendents)
- total ms/call: average time per execution (including descendents)

GPROF (2)

- symtablehash.c

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
85.71	0.36	0.36	457755	0.00	0.00	SymTable_put
7.14	0.39	0.03	495194	0.00	0.00	SymTable_hash
2.38	0.40	0.01	11020	0.00	0.00	SymTable_get
2.38	0.41	0.01	10010	0.00	0.04	SymTable_remove
2.38	0.42	0.01	6	1.67	1.83	SymTable_expand

- name: name of function
- %time: %age of time spent executing this function
- cum. secs: [not relevant]
- self seconds: time spent executing this function
- calls: number of times function was called (excluding recursion)
- self ms/call: avg time per execution (excluding descendents)
- total ms/call: average time per execution (including descendents)

GPROF (2)

- Call graph profile
  - Each section describes one function
    - Which functions called it, time consumed? *etc.*
    - Which functions it calls, how many times, and for how long? *etc.*
- Usually overkill; we won't look at this output in any detail

GPROF (2)

- Observation:
  - symtablelist is investing way too much time in SymTable_eexists() as was expected.
  - symtablehash mitigates the problem… again as expected

Algorithms & Data Structures (3)

- Use a better algorithm or data structure
  - *e.g.*: <u>hashtables</u> over <u>linked-lists</u>
- Depends on
  - Data
  - Hardware
  - OS *etc.*

Compiler Speed Optimization (4)

- Enable compiler speed optimization
  `$ gcc209 -Ox sample.c -o sample`
  - Compilation time increases
  - Speeds up execution
  - **x** can be 1, 2, or 3
- See "man gcc" for details
- Optimization modifies symbol table
  - gdb cannot identify variables' data during debugging sessions

Code Tuning (5)

```
for (i = 0; i < strlen(s); i++) {
   /* do something with s[i] here */
}
```

```
length = strlen(s);
for (i = 0; i < length; i++) {
   /* do something with s[i] here */
}
```

Code Tuning (5)

ACHTUNG!

- **Can introduce redundant code**
- **Some compilers support inline keyword**

```
void g(void) {
   /* some code */
}
void f(void) {
   ...
   g();
   ...
}
```

```
void f(void) {
   ...
   /* same code here */
   ...
}
```

Code Tuning (5)

LOOP UNROLLING

```
for (i = 0; i < 6; i++)
   a[i] = b[i] + c[i];
```

```
for (i = 0; i < 6; i += 2) {
   a[i] = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
}
```

```
a[i] = b[i] + c[i];
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
a[i+3] = b[i+3] + c[i+3];
a[i+4] = b[i+4] + c[i+4];
a[i+5] = b[i+5] + c[i+5];
```

SOME COMPILERS PROVIDE `-funroll-loops` OPTION

Code Tuning (5)

- Rewrite key functions in low-level language *e.g.* assembly language
  - Use registers

- Beware: modern optimizing compilers generate fast code
  - Hand-written assembly language code could be slower than compiler-generated code, especially when compiled with optimization flag

Improving Memory Efficiency

- Memory is inexpensive

- SPACE less relevant than TIME

Improving Memory Efficiency

- Use a smaller data type
  - *e.g.* `short` instead of `int`

- Enable compiler <u>size</u> optimization

```
$ gcc209 -Os sample.c -o sample
```

Summary

- Clarity supersedes performance

Don’t improve performance unless you must!!!