Building and Performance

Jennifer Rexford

Goals of Part 1 (Building)

- 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`**

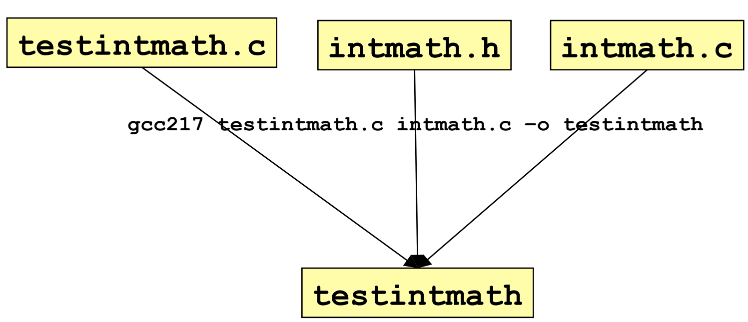
Example of a Three-File Program

- 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**

3

Motivation for Make (Part 1)

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



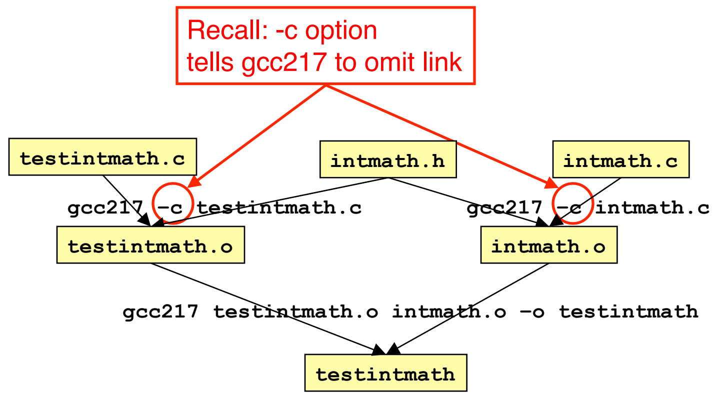
That's not how it's done in the real world…

4

Motivation for Make (Part 2)

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

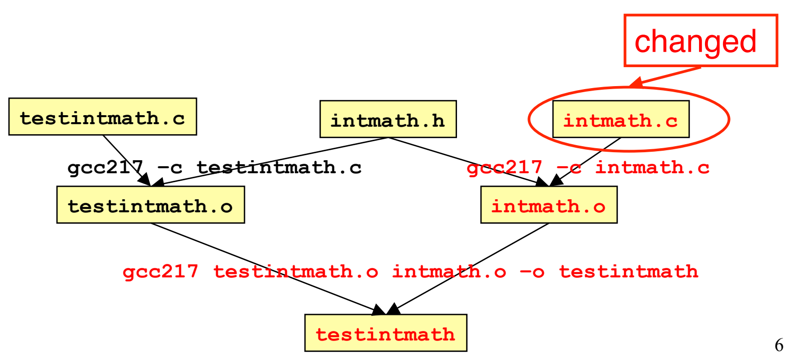
That's how it's done in the real world; Why?...



5

Partial Builds

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

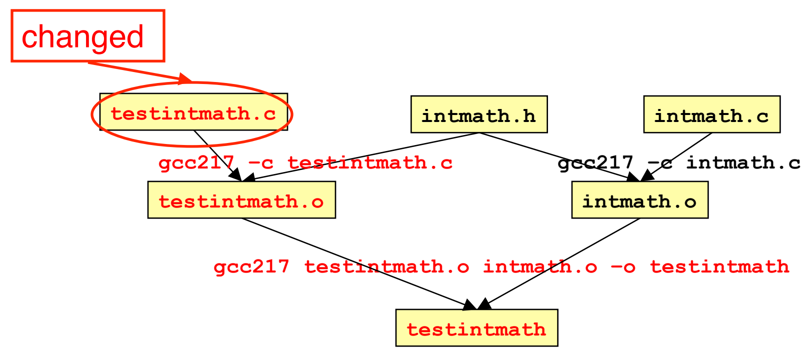


6

Partial Builds (cont.)

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

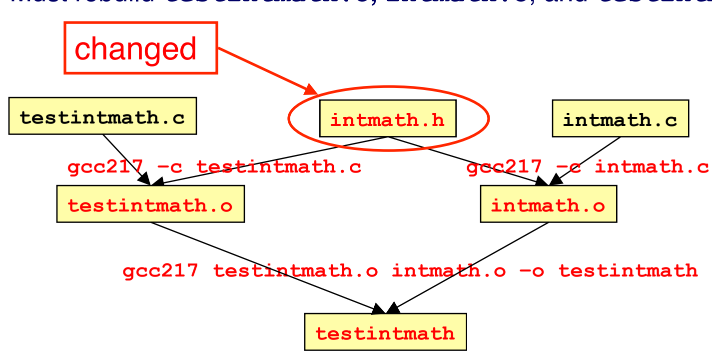
If program contains many .c files, could save many hours of build time



7

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**



8

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`**!

9

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***

10

Dependency Rules

- Dependency rule syntax

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

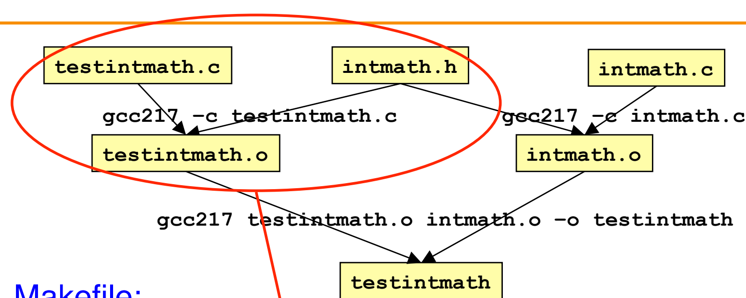
  - **target**: the file you want to build
  - **dependencies**: the files on which the target depends
  - **command**: what to execute to create the target (after a TAB character)

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

- Work recursively; examples illustrate…

11

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

12

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

13

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
- 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"

```
all: testintmath

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

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

14

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
```

15

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
```

Same idea here, but "clobber" depends upon "clean"

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

"all" depends upon "testintmath"

```
$ make all
gcc217 -c testintmath.c
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath
```

"all" is the default target

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

16

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
  ```

17

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
```

18

Version 3 in Action

- Same as Version 2

19

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

- See Appendix for 2 additional versions

20

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

21

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

22

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

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

25

Appendix: Fancy Stuff

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

26

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 $@
```

27

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 $<
```

28

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
```

Appendix: Version 5 in Action

- Same as Version 2

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

Performance Improvement

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 7

35

Goals of Part 2 (Performance)

- Help you learn about:
  - Techniques for improving program performance
    - How to make your programs run faster and/or use less memory
  - The GPROF execution profiler
- Why?
  - In a large program, typically a small fragment of the code consumes most of the CPU time and/or memory
  - A power programmer knows how to identify such code fragments
  - A power programmer knows techniques for improving the performance of such code fragments

36

Performance Improvement Pros

- Techniques described in this lecture can yield answers to questions such as:
  - 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?

37

Performance Improvement Cons

- Techniques described in this lecture can yield code that:
  - Is less clear/maintainable
  - Might confuse debuggers
  - Might contain bugs
    - Requires regression testing
- So…

38

When to Improve Performance

"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

39

Execution Efficiency

- We propose 5 steps to improve execution (time) efficiency
- Let's consider one at a time…

40

Timing Studies

(1) Do timing studies

- To time a program… Run a tool to time program execution
  - E.g., Unix **time** command

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

- Output:
  - **Real**: Wall-clock time between program invocation and termination
  - **User**: CPU time spent executing the program
  - **System**: CPU time spent within the OS on the program's behalf
- But, which *parts* of the code are the most time consuming?

41

Timing Studies (cont.)

- To time *parts of* a program... Call a function to compute **wall-clock time** consumed
  - E.g., Unix **gettimeofday()** function (time since Jan 1, 1970)

```
#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);
```

  - Not defined by C90 standard

42

Timing Studies (cont.)

- To time *parts of* a program... Call a function to compute **CPU time** consumed
  - E.g. **clock()** function

```
#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;
```

  - Defined by C90 standard

Identify Hot Spots

(2) Identify hot spots

- Gather statistics about your program's execution
  - How much time did execution of a function take?
  - How many times was a particular function called?
  - How many times was a particular line of code executed?
  - Which lines of code used the most time?
  - Etc.
- How? Use an **execution profiler**
  - Example: **gprof** (GNU Performance Profiler)

GPROF Example Program

- Example program for GPROF analysis
  - Sort an array of 10 million random integers
  - Artificial: consumes much CPU time, generates no output

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

enum {MAX_SIZE = 10000000};
int a[MAX_SIZE]; /* Too big to fit in stack! */

void fillArray(int a[], int size) {
   int i;
   for (i = 0; i < size; i++)
      a[i] = rand();
}

void swap(int a[], int i, int j) {
   int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
…
```


GPROF Example Program (cont.)

- Example program for GPROF analysis (cont.)

```
…
int partition(int a[], int left, int right) {
   int first = left-1;
   int last = right;
   for (;;) {
      while (a[++first] < a[right])
         ;
      while (a[right] < a[--last])
         if (last == left)
            break;
      if (first >= last)
         break;
      swap(a, first, last);
   }
   swap(a, first, right);
   return first;
}
…
```

GPROF Example Program (cont.)

- Example program for GPROF analysis (cont.)

```
…
void quicksort(int a[], int left, int right) {
   if (right > left)
   {
      int mid = partition(a, left, right);
      quicksort(a, left, mid - 1);
      quicksort(a, mid + 1, right);
   }
}

int main(void) {
   fillArray(a, MAX_SIZE);
   quicksort(a, 0, MAX_SIZE - 1);
   return 0;
}
```

47

Using GPROF

- Step 1: Instrument the program

  `gcc217 -pg mysort.c -o mysort`
  - Adds profiling code to mysort, that is…
  - “Instruments” `mysort`
- Step 2: Run the program

  `mysort`
  - Creates file `gmon.out` containing statistics
- Step 3: Create a report

  `gprof mysort > myreport`
  - Uses `mysort` and `gmon.out` to create textual report
- Step 4: Examine the report

  `cat myreport`

48

The GPROF Report

- Flat profile

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 84.54      2.27     2.27  6665307    0.00     0.00  partition
  9.33      2.53     0.25 54328749    0.00     0.00  swap
  2.99      2.61     0.08        1    0.08     2.61  quicksort
  2.61      2.68     0.07        1    0.07     0.07  fillArray
```

- Each line describes one function
  - name: name of the function
  - %time: percentage of time spent executing this function
  - cumulative seconds: [skipping, as this isn't all that useful]
  - self seconds: time spent executing this function
  - calls: number of times function was called (excluding recursive)
  - self s/call: average time per execution (excluding descendents)
  - total s/call: average time per execution (including descendents)

The GPROF Report (cont.)

- Call graph profile

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    2.68                 main [1]
                0.08    2.53       1/1           quicksort [2]
                0.07    0.00       1/1           fillArray [5]
-----------------------------------------------
                             13330614             quicksort [2]
                0.08    2.53       1/1           main [1]
[2]     97.4    0.08    2.53       1+13330614 quicksort [2]
                2.27    0.25 6665307/6665307      partition [3]
                             13330614             quicksort [2]
-----------------------------------------------
                2.27    0.25 6665307/6665307     quicksort [2]
[3]     94.4    2.27    0.25 6665307         partition [3]
                0.25    0.00 54328749/54328749     swap [4]
-----------------------------------------------
                0.25    0.00 54328749/54328749     partition [3]
[4]      9.4    0.25    0.00 54328749          swap [4]
-----------------------------------------------
                0.07    0.00       1/1           main [1]
[5]      2.6    0.07    0.00       1         fillArray [5]
-----------------------------------------------
```

The GPROF Report (cont.)

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

51

GPROF Report Analysis

- Observations
  - **`swap()`** is called very many times; each call consumes little time; **`swap()`** consumes only 9% of the time overall
  - **`partition()`** is called many times; each call consumes little time; but **`partition()`** consumes 85% of the time overall
- Conclusions
  - To improve performance, try to make **`partition()`** faster
  - Don't even think about trying to make **`fillArray()`** or **`quicksort()`** faster

52

GPROF Design

- Incidentally…
- How does GPROF work?
  - Good question!
  - Essentially, by randomly sampling the code as it runs
  - … and seeing what line is running, & what function it's in

53

Algorithms and Data Structures

(3) Use a better algorithm or data structure

- Example:
  - For mysort, would mergesort work better than quicksort?
- Depends upon:
  - Data
  - Hardware
  - Operating system
  - …

54

Compiler Speed Optimization

(4) Enable compiler speed optimization

```
gcc217 -Ox mysort.c -o mysort
```

- Compiler spends more time compiling your code so…
- Your code spends less time executing
- **x** can be:
  - 1: optimize
  - 2: optimize more
  - 3: optimize yet more
- See "man gcc" for details

- Beware: Speed optimization can affect debugging
  - E.g. Optimization eliminates variable => GDB cannot print value of variable

55

Tune the Code

(5) Tune the code

- Some common techniques
  - **Factor** computation out of loops

- Example:

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

- Faster:

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

56

Tune the Code (cont.)

- Some common techniques (cont.)
  - **Inline** function calls
    - Example:

```
void g(void) {
   /* Some code */
}
void f(void) {
   …
   g();
   …
}
```

    - Maybe faster:

```
void f(void) {
   …
   /* Some code */
   …
}
```

    - Beware: Can introduce redundant/cloned code
    - Some compilers support **`inline`** keyword

57

Tune the Code (cont.)

- Some common techniques (cont.)
  - **Unroll** loops
    - Example:

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

    - Maybe faster:

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

    - Maybe even faster:

```
a[i+0] = b[i+0] + c[i+0];
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 option, e.g. **`-funroll-loops`**

58

Tune the Code (cont.)

- Some common techniques (cont.):
  - Rewrite in a lower-level language
    - Write key functions in **assembly language** instead of C
      - Use registers instead of memory
      - Use instructions (e.g. `adc`) that compiler doesn't know
    - Beware: Modern optimizing compilers generate fast code
      - Hand-written assembly language code could be *slower* than compiler-generated code, especially when compiled with speed optimization

59

Execution Efficiency Summary

- Steps to improve execution (time) efficiency:
  (1) Do timing studies
  (2) Identify hot spots
  (3) Use a better algorithm or data structure
  (4) Enable compiler speed optimization
  (5) Tune the code

60

Improving Memory Efficiency

- These days, memory is cheap, so…
- Memory (space) efficiency typically is less important than execution (time) efficiency
- Techniques to improve memory (space) efficiency…

61

Improving Memory Efficiency

(1) Use a smaller data type

- E.g. `short` instead of `int`

(2) Compute instead of storing

- E.g. To determine linked list length, traverse nodes instead of storing node count

(3) Enable compiler ***size*** optimization

```
gcc217 -Os mysort.c –o mysort
```

62

Summary

- Steps to improve execution (time) efficiency:
  - (1) Do timing studies
  - (2) Identify hot spots *
  - (3) Use a better algorithm or data structure
  - (4) Enable compiler speed optimization
  - (5) Tune the code
  - * Use GPROF
- Techniques to improve memory (space) efficiency:
  - (1) Use a smaller data type
  - (2) Compute instead of storing
  - (3) Enable compiler size optimization
- And, most importantly…

63

Summary (cont.)

Clarity supersedes performance

Don't improve performance unless you must!!!

64