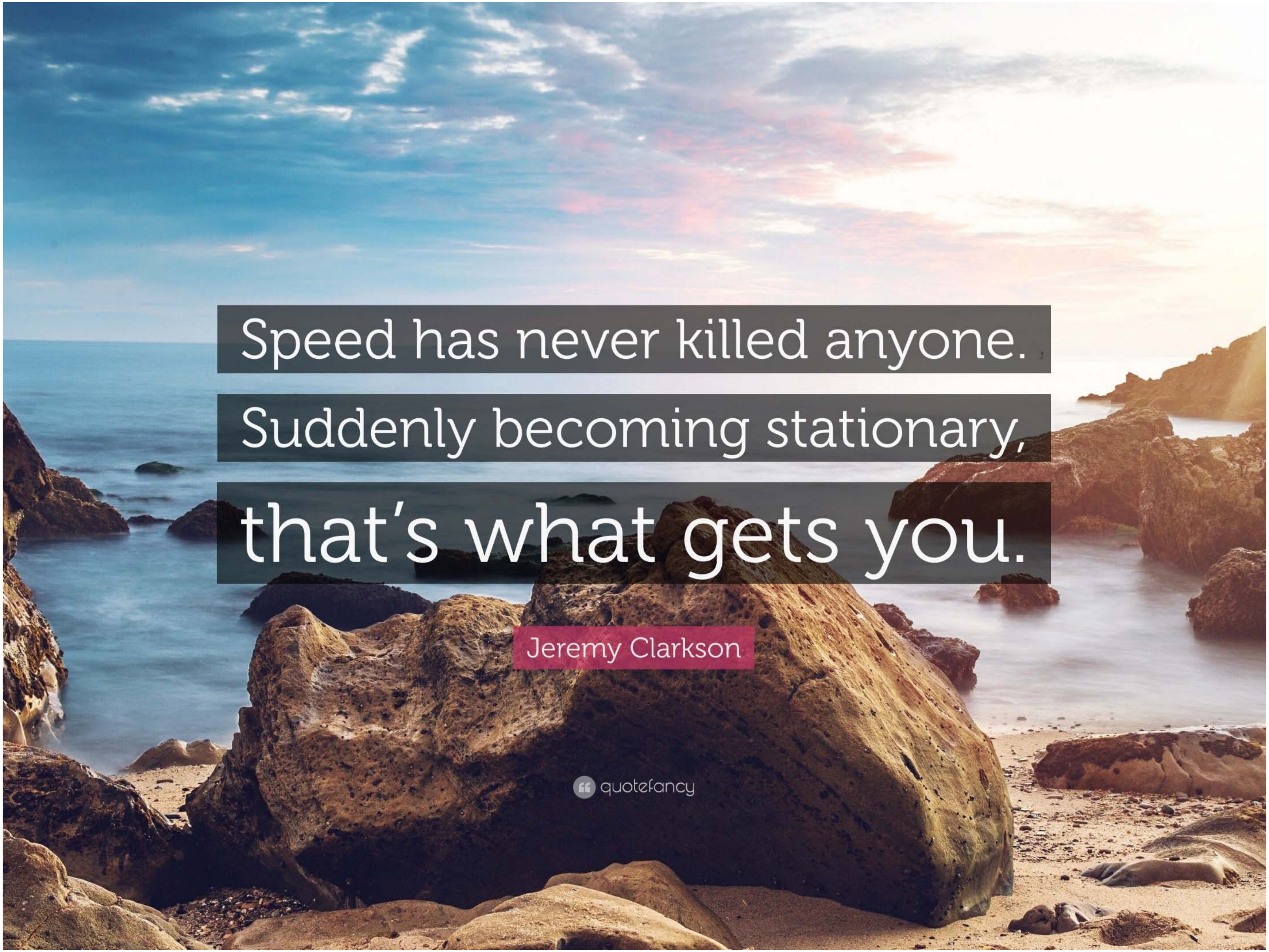
Applied Programming

Program Performance Evaluation

Execution Time


Speed has never killed anyone.
Suddenly becoming stationary,
that's what gets you.
Jeremy Clarkson
quotefancy

Performance Evaluation

- The most common way to evaluate the performance of a program are:
    1. **Execution time**
    2. **Memory use**
- It is difficult to measure accurately time and memory because often:
  - Methods are too intrusive
  - Tools used are too course-grained

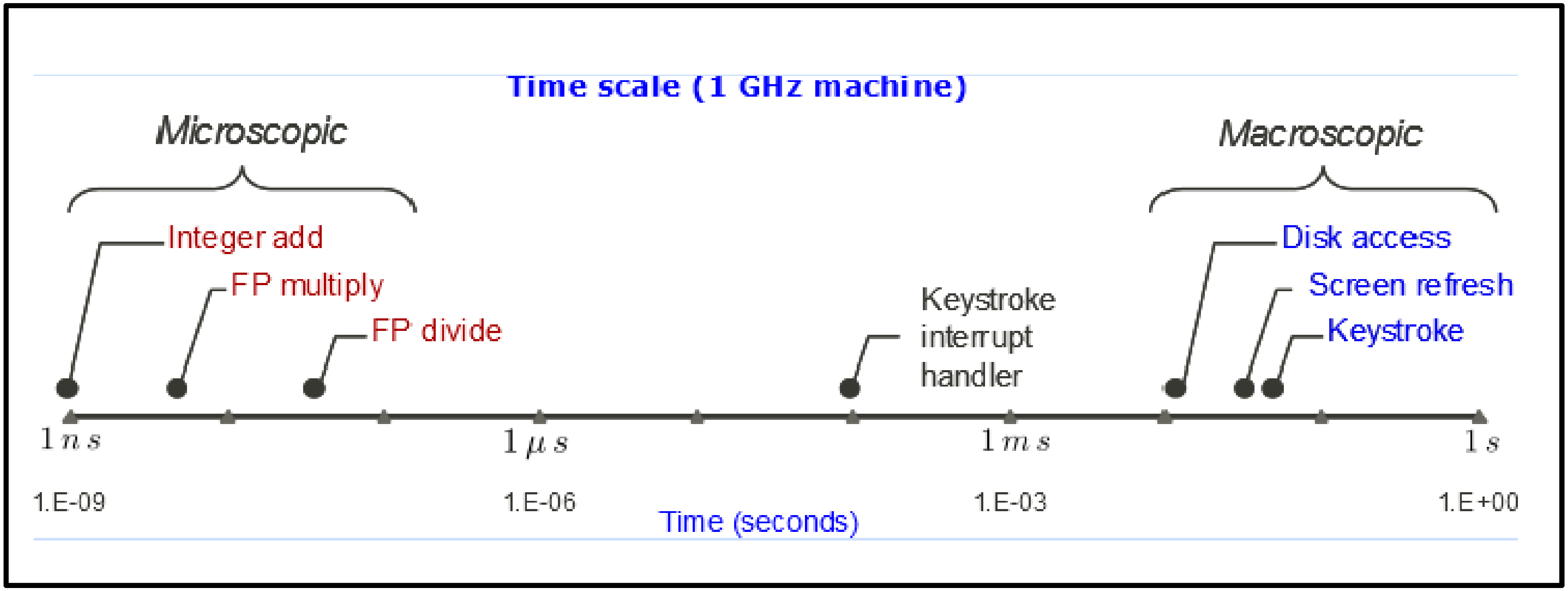
More details in: G. Semeraro, “Numerical Programming in C”, chapter 2, pp 22-28 (in Mycourses)

Performance: Execution Time

- C includes (**`<time.h>`**) functions and typemarks that can be used to measure both ***relative and absolute execution time***

- We are interested in relative time, (how long it takes a program to execute) as a performance metric
  - The only measure of program performance that we will explore here is **execution time**.

Implementing Timing



- **Keep in mind the time scale of your measurements.**
- **If you do not have a timer with the desired resolution you will need to repeat the segment of code to be timed.**

How to “time” Programs

- **Basic approach to determine execution time of a section of code:**
  1. **Record start time**
  2. **Loop {Execute code (to be timed)}**
  3. **Record stop time**
  4. **Determine time elapsed (stop -start times)/loops**

- **Need to add code for**
  a. **Recording “time”**
  b. **Performing arithmetic on “time values”**

Timing Trouble

- We often need to loop our code to be measured
  - Our timing accuracy is low (ms range)
  - Our computers are fast (ns range)
- Need to be careful that looping the code does not alter its behavior
  - File access
  - Memory allocation

www.flickr.com/photos/baderadam/7464107388

Timing Trouble Example

- Consider the following measurement pseudo code:

```
        Open(file)
loop{
        read(file)
        process()  }
        Close(file)
```

Red is our Additional Measurement code

- Is there an issue here?
  - The read operation will only happen ONCE because files are read sequentially.

Timing Trouble Resolution

- Consider the following measurement pseudo code:

```
loop{  Open(file)
       read(file)
       process()
       Close(file)  }
```

Red is our Additional Measurement code

- By opening and closing the file inside the loop we force the file system to reset.
  - Often we have to restructure the code to measure it!

Or we could use fseek() to reset the read file handle

More Timing Trouble

- Consider the following code:

  loop{ Open(file) ; read(file); process(); Close(file) }

  - We want to improve "process()"
  - How much does "read(file)" affect us?

- A LOT, often enough to **cloud** (pun intended) our improvements
  - We have to know "where" our I/O is.
    - Local HD?
    - Cloud (networked) HDD?

- IDEA: Feed **process()** dummy, non-HDD drive data

Recording Time

- C functions for ***reading the internal clock*** are in **<time.h>**
- To declare a time variable:

```
clock_t time_start
```

- Function prototype:

```
clock_t clock(void);
```

  - Returns **"CPU clock time"**
  - In "CPU clock ticks" not seconds

Code Timing Example

```
#include <time.h>
. . .
clock_t StartTime,StopTime,ExecutionTime;
. . .
StartTime = clock (); /* Record start */
/* Code to be timed goes here
 *. . .
 */
StopTime = clock ();  /* Record end */
ExecutionTime = ( StopTime – StartTime);/* Compute elapsed */
```

- To print out the execution time we need to know the type for `clock_t`
  - We need to know the "unit of time"
  - Defined by `CLOCKS_PER_SEC`

Code Timing Example…

```
#include <time.h>
. . .
clock_t CPUTimeExec, /* Time Elapsed          */
        CPUTimeStart,/* Start “stopwatch”     */
. . .
CPUTimeStart = clock();

/* Code to be timed goes here */
CPUTimeExec = (CPUTimeStart – clock());
printf (“Execution time = %f s\n”,
 (double) CPUTimeExec /(double) CLOCKS_PER_SEC);
```

Timing1.c (full code)

```
/* Example: Simple code timing1.c */
#include <stdio.h>
#include <time.h>
#define REPEAT 10000000L
int main () {
  int         A, B, C;
  long        Index;
  clock_t     StartTime, StopTime, ElapsedTime;
  A=15; B=3;                                    /* Initialize Variables */

  StartTime = clock();                          /* Timing starts */
    for (Index = 0; Index < REPEAT; Index++) { C=A*A/B; }
    StopTime = clock();
  ElapsedTime=  StopTime - StartTime;           /* Timing ends */

/* Show Results */
  printf ("Operation %d^2/%d repeated %ld times\n",A,B,REPEAT);
  printf ("** CLOCKS_PER_SEC  is %ld ticks/sec\n", CLOCKS_PER_SEC);
  printf ("** Size of clock_t is %d bytes\n", (int) sizeof(clock_t));
  printf ("   Total Time = %ld \"clocks ticks\"\n", ElapsedTime);
  printf ("   Total Time = %f [sec]\n", (double) ElapsedTime / (double) CLOCKS_PER_SEC);
  printf ("   Time per Iteration=%4.2g [sec]\n",
                        (double)ElapsedTime/(double)CLOCKS_PER_SEC/REPEAT);
return 0; }
```

Timing1.c (parts we care about)

```
StartTime = clock();
for (Index = 0; Index < REPEAT; Index++) { C=A*A/B; }
StopTime = clock();
ElapsedTime=  StopTime – StartTime;
printf ("Operation %d^2/%d repeated %ld times\n",A,B,REPEAT);
printf ("** CLOCKS_PER_SEC  is %ld ticks/sec\n", CLOCKS_PER_SEC);
printf ("** Size of clock_t is %d bytes\n", (int) sizeof(clock_t));
printf ("   Total Time = %ld \"clocks ticks\"\n", ElapsedTime);
printf ("   Total Time = %f [sec]\n",
              (double) ElapsedTime / (double) CLOCKS_PER_SEC);
printf ("   Time per Iteration=%4.2g [sec]\n",
              (double) ElapsedTime / (double) CLOCKS_PER_SEC/REPEAT);
```

```
Operation 15^2/3 repeated 10,000,000 times
** CLOCKS_PER_SEC  is 1,000,000 ticks/sec
** Size of clock_t is 8 bytes
   Total Time = 60,000 "clocks ticks"
   Total Time = 0.060000 [sec]
   Time per Iteration=6e-09 [sec]
```

Conditional timing code

- Measuring timing performance takes extra code and impacts the code execution
  - Want to have an “easy” way to measure performance AND not impact production code

- Add 2 sections of code, one with timing, and one without and then use #define features to switch!
  - Use the gcc –Dxxxx feature, where xxxx is the name of your #define variable

Timing2.c (1/2)

```
/*  Example: Simple code timing,  Uses Conditional Compilation EN_TIME */
#include <stdio.h>
#include <time.h>

#ifdef EN_TIME
 #define REPEAT 10000000L
#else
 #define REPEAT 1
#endif

int main () {
  int                    A, B, C;
  long                   Index;
#ifdef EN_TIME
  clock_t    StartTime, StopTime, ElapsedTime;
#endif

  /* Initialize Variables */
  A=15; B=3;

  /* Timing starts */
#ifdef EN_TIME
  StartTime = clock();
#endif
```

This code takes advantage of the gcc –D feature

```
gcc -ansi –DEN_TIME timing.c
vs
gcc -ansi timing.c
```

Timing2.c (2/2)

```
for (Index = 0; Index < REPEAT; Index++) {
   C=A*A/B;
  }
#ifdef EN_TIME
  StopTime = clock();
  ElapsedTime=  StopTime - StartTime;
#endif
  /* Timing ends */
  /* Show Results */
#ifdef EN_TIME
  printf ("Operation %d=%d^2/%d repeated %ld times\n",C,A,B,REPEAT);
#else
  printf ("Operation %d=%d^2/%d repeated %d times\n",C,A,B,REPEAT);
#endif
  printf ("** CLOCKS_PER_SEC  is %ld ticks/sec\n", CLOCKS_PER_SEC);
#ifdef EN_TIME
  printf ("** Size of clock_t is %d bytes\n", (int) sizeof(clock_t));
  printf ("   Total Time = %ld \"clocks ticks\"\n", ElapsedTime);
  printf ("   Total Time = %f [sec]\n", (double) ElapsedTime / (double) CLOCKS_PER_SEC);
  printf ("   Time per Iteration=%4.2g [sec]\n",
                              (double) ElapsedTime / (double) CLOCKS_PER_SEC/REPEAT);
#endif
return 0;
}
```

gcc –Dxxx example

```
#ifdef EN_TIME
  printf ("Operation %d=%d^2/%d repeated %ld times\n",C,A,B,REPEAT);
#else
  printf ("Operation %d=%d^2/%d repeated %d times\n",C,A,B,REPEAT);
#endif
  printf ("** CLOCKS_PER_SEC  is %ld ticks/sec\n", CLOCKS_PER_SEC);
#ifdef EN_TIME
  printf ("** Size of clock_t is %d bytes\n", (int) sizeof(clock_t));
  printf ("   Total Time = %ld \"clocks ticks\"\n", ElapsedTime);
  printf ("   Total Time = %f [sec]\n", (double) ElapsedTime / (double) CLOCKS_PER_SEC);
  printf ("   Time per Iteration=%4.2g [sec]\n",
                                    (double) ElapsedTime / (double) CLOCKS_PER_SEC/REPEAT);
#endif
```

```
gcc –ansi –DEN_TIME timing.c –o timing2
Operation 75=15^2/3 repeated 10000000 times
** CLOCKS_PER_SEC  is 1000000 ticks/sec
** Size of clock_t is 8 bytes
   Total Time = 60000 "clocks ticks"
   Total Time = 0.060000 [sec]
   Time per Iteration=6e-09 [sec]
gcc –ansi timing.c –o timing2
Operation 75=15^2/3 repeated 1 times
** CLOCKS_PER_SEC  is 1000000 ticks/sec
```

Timing with Macros

- Need to ***instrument the code for timing***
  - in such a way that timing code can be easily "removed" from the final release code.

- We will ***use Macros together with conditional compilation***
  - inline the timing code (instrumentation) only for **"testing"** and leave it out for **"production''**.

Timing with Macros

- Organize all **timing macros** in **`Timers.h`**

- Include the header **`Timers.h`** in the program to be instrumented.
- Use the macros in **`Timers.h`** where necessary to insert timing code.

- Use conditional compilation to ***include timing code only when desired,*** e.g., use **`-DEN_TIME`** to **`EN`**able **`TIME`** code for inclusion.
  - *If **`EN_TIME`** is not defined the macros expand to "nothing"*

Reference: G. Semeraro, "Numerical Programming in C", chapter 2, pp 32-39 (available in Mycourses)

Timers.h

- Provides the following macros:
  - **DECLARE_REPEAT_VAR(V)**
  - **DECLARE_TIMER(A)**
  - **BEGIN_REPEAT_TIMING(R,V)**
  - **END_REPEAT_TIMING**
  - **START_TIMER(A)**
  - **RESET_TIMER(A)**
  - **STOP_TIMER(A)**
  - **PRINT_TIMER(A)**
  - **PRINT_RTIMER(A,R)**

Reference: For a detailed example of how to use it see G. Semeraro's book, chapter 2,

DECLARE_TIMER(A)

```
#define DECLARE_TIMER(A)                                                   \
    struct timmerDetails {                                                 \
      /* Start Time - set when the timer is started */                     \
      clock_t Start;                                                       \
      /* Stop Time - set when the timer is stopped */                      \
      clock_t Stop;                                                        \
      /* Elapsed Time - Accumulated when the timer is stopped */           \
      clock_t Elapsed;                                                     \
      /* Timer State - Set automatically: 0=stopped / 1=running */         \
      int State;                                                           \
    } A = { /* Elapsed Time and State must be initialized to zero */ \
      /* Start   = */ 0,                                                   \
      /* Stop    = */ 0,                                                   \
      /* Elapsed = */ 0,                                                   \
      /* State   = */ 0,                                                   \
    }; /* Timer has been declared and defined */
```

- Normally needs to be in global scope
  - Because you will want to print the results

START_TIMER(A)

```
#define START_TIMER(A)                                                    \
    {                                                                     \
     /* It is an error if the timer is currently running */               \
     if (1 == A.State)  {                                                 \
       fprintf(stderr, "Error, running timer "#A" started.\n");           \
       }                                                                  \
     /* Set the state to running */                                       \
     A.State = 1;                                                         \
     /* Set the start time, done last to maximize accuracy */             \
     A.Start = clock();                                                   \
    } /* START_TIMER() */
```

- Will report an error if already started
- Records the starting time stamp

RESET_TIMER(A)

```
#define RESET_TIMER(A)                                              \
    {                                                               \
     /* Reset the elapsed time to zero */                           \
     A.Elapsed = 0;                                                 \
    } /* RESET_TIMER() */
```

- Just resets the elapsed time value
- Does not “stop” any running time

STOP_TIMER(A)

```
#define STOP_TIMER(A)                                                      \
    {                                                                      \
     /* Set the stop time, done first to maximize accuracy */             \
     A.Stop = clock();                                                     \
     /* It is an error if the timer is currently stopped */               \
     if (0 == A.State) {                                                   \
       fprintf(stderr, "Error, stopped timer "#A" stopped again.\n");\
       }                                                                   \
     else {/*accumulate running total only if previously running */      \
       A.Elapsed += A.Stop - A.Start;                                      \
       }                                                                   \
     /* Set the state to stopped */                                       \
     A.State = 0;                                                          \
    } /* STOP_TIMER() */
```

- Sets the stop time
  - Prints an error message if the timer already stopped
  - Accumulate the elapsed time
- Sets the state to stopped.

PRINT_TIMER(A)

```
#define PRINT_TIMER(A)                                                \
    {                                                                  \
     /* Stop the timer (silently) if it is currently running */      \
     if (1 == A.State)  {                                              \
       STOP_TIMER(A);    /* no error possible in this case */        \
       }                                                               \
     fprintf(stderr, "Elapsed CPU Time ("#A") = %g sec.\n",           \
                     (double)A.Elapsed / (double)CLOCKS_PER_SEC);    \
    } /*PRINT_TIMER() */
```

- Stops the timer, if still running
- Prints the elapsed time in seconds to stderr

PRINT_RTIMER(A,R)

```
#define PRINT_TIMER(A)                                                    \
    {                                                                     \
     /* Stop the timer (silently) if it is currently running */          \
     if (1 == A.State)  {                                                 \
       STOP_TIMER(A);    /* no error possible in this case */            \
       }                                                                  \
     fprintf(stderr, "Elapsed CPU Time per Iteration ("#A", %d) = %  \
            .2e sec.\n", R, ((double)A.Elapsed /                         \
     (double)CLOCKS_PER_SEC)/(double)R);  \
    } /*PRINT_RTIMER() */
```

- Works just print timer but takes into account the number of timing loops

DECLARE_REPEAT_VAR(V)

- Will need to allocate UNIQUE variables outside the scope of functions
  - See the notes on "**Using ## to create variables**" at the top of this document.

- A task for the student to implement

Repeat support

```
▪BEGIN_REPEAT_TIMING(R,V)
▪END_REPEAT_TIMING
```

- Some macros must be implemented as pairs
  - See the notes on "**Macros outside the box**" at the top of this document.

- A task for the student to implement

Testing Timer Macros

- How do you know if your timer macros return good data?
  - Time something you know

```
#include <unistd.h>
int main() {
   clock_t end_t;
   int delay;
   /* Your macro stuff here */
   end_t = clock() + 60 * CLOCKS_PER_SEC;
   while (end_t > clock()) /* wait 60 secs */
      {/* Consume CPU time */
      delay = 1<<19;
      while (delay) {delay--;}
      }
   /* more of your macro stuff */
   return 0;
}
```

- Always write testers to verify any code or macros you develop. Never ASSUME you code works.

Compiling with **`Timers.h`**

Example: **`Timing.c`** (show example)

- To compile with timing

```
gcc –Wall –pedantic –ansi –DEN_TIMING Timing.c –o Timing
```

file

- To compile without timing

```
gcc –Wall –pedantic –ansi Timing.c –o NoTiming
```

- To generate preprocessor output (for macro debugging)

```
gcc –DEN_TIME –E –P Timing.c > ExpandedCode.c
```

Timing measurement with `time`

- Example: Timing with GNU/Linux **`time`**

```
/usr/bin/time -p ./Timing
```

- Used to report the Linux view of overall program usage
  - Doesn't provide specific details

FYI: The **bash** shell also has a function called **`time`**. To call the correct **`time`** function you may need to specify the full path to it,
e.g., **`/usr/bin/time`**

try: which time to verify you get the right one

The GNU/Linux `time`

- The GNU/Linux function **`time`** (**`/usr/bin/time`**) reports ***three timing measurements***

  1. ***Real Time (RT):*** Actual time elapsed as measured by an external clock (*e.g.*, your watch)

  2. ***User Time (UT):*** Amount of time the ***program is actually executing*** (programs spend some of their "real time" waiting to be executed)

  3. ***System Time (ST):*** Amount of time spent executing operating systems code *on the program's behalf*

     Note: $RT \geq UT + ST$

Timing example

```
/usr/bin/time -p ./Timing
```

Elapsed CPU Time (MainTimer) = 13.29 sec.

real 13.31

user 13.29

sys 0.00

Blue text is our code, black is the Linux time function.

Real (wall) time is longer, due to the time required to load our program.

User time matches our MainTimer because we didn’t get swapped out by the OS.

Our code didn’t make any OS calls so there is no system time

Timing: Concluding Remarks

- Timing reported by instrumentation code includes both ***"user time" and "system time"***

- By default the programs are ***linked dynamically*** by **`gcc`**. This can ***affect timing.***

- It may be worth to assess timing with ***statically linked*** libraries, for that use:

```
gcc –static-libgcc …
```

Timing: Concluding Remarks

- Accuracy of timing ***depends*** also ***on the implementation*** of **`clock()`**

- Often the ***resolution*** of **`clock()`** is ***not sufficient***, so you ***need to iterate*** on the portion of code being timed.

  Price to pay (no free lunch)

  - ***Iterations decrease accuracy*** of timing (due among other factors to cache). Use iterations only when strictly necessary

Applied Programming

Program Performance Evaluation

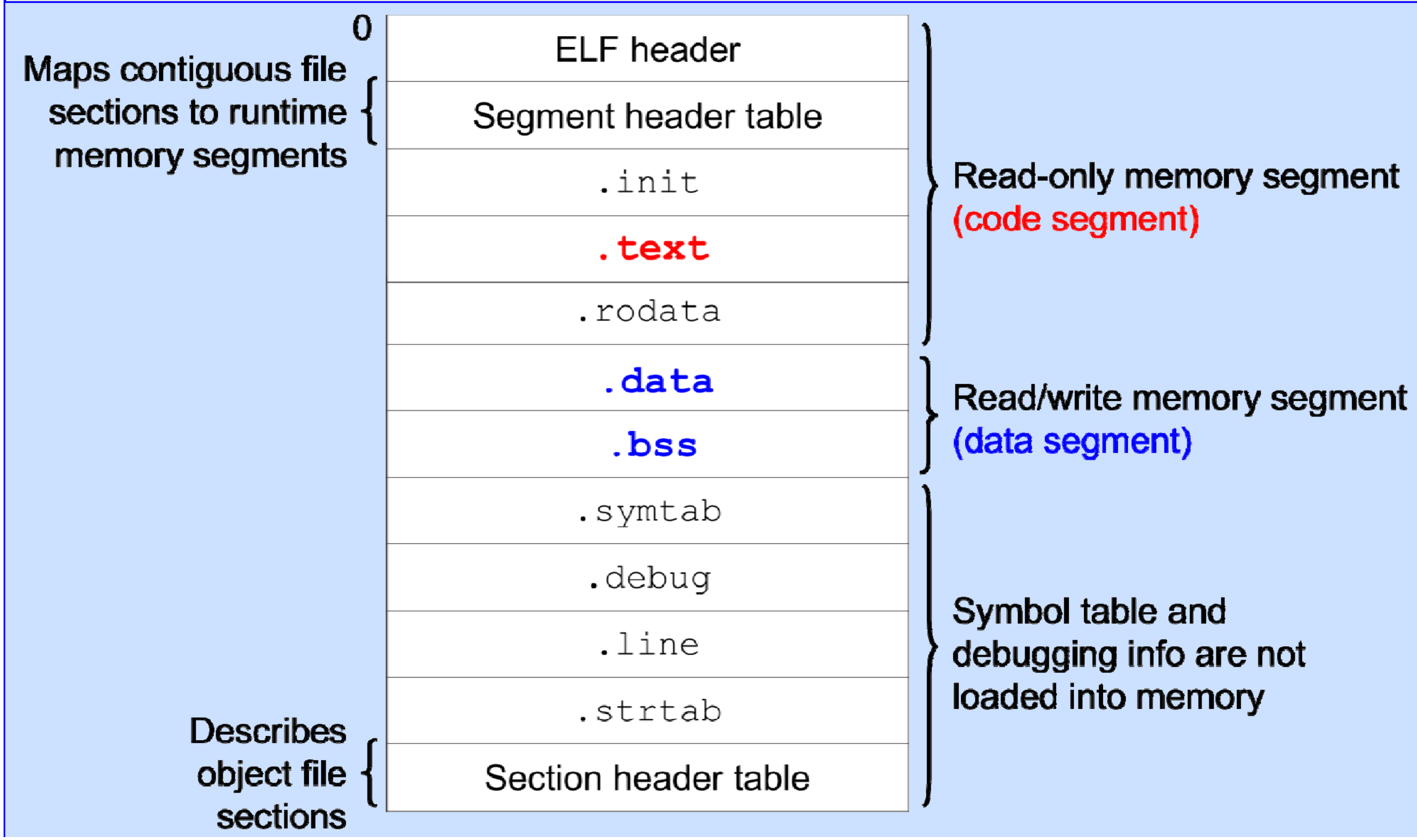
Memory Use

Memory Use

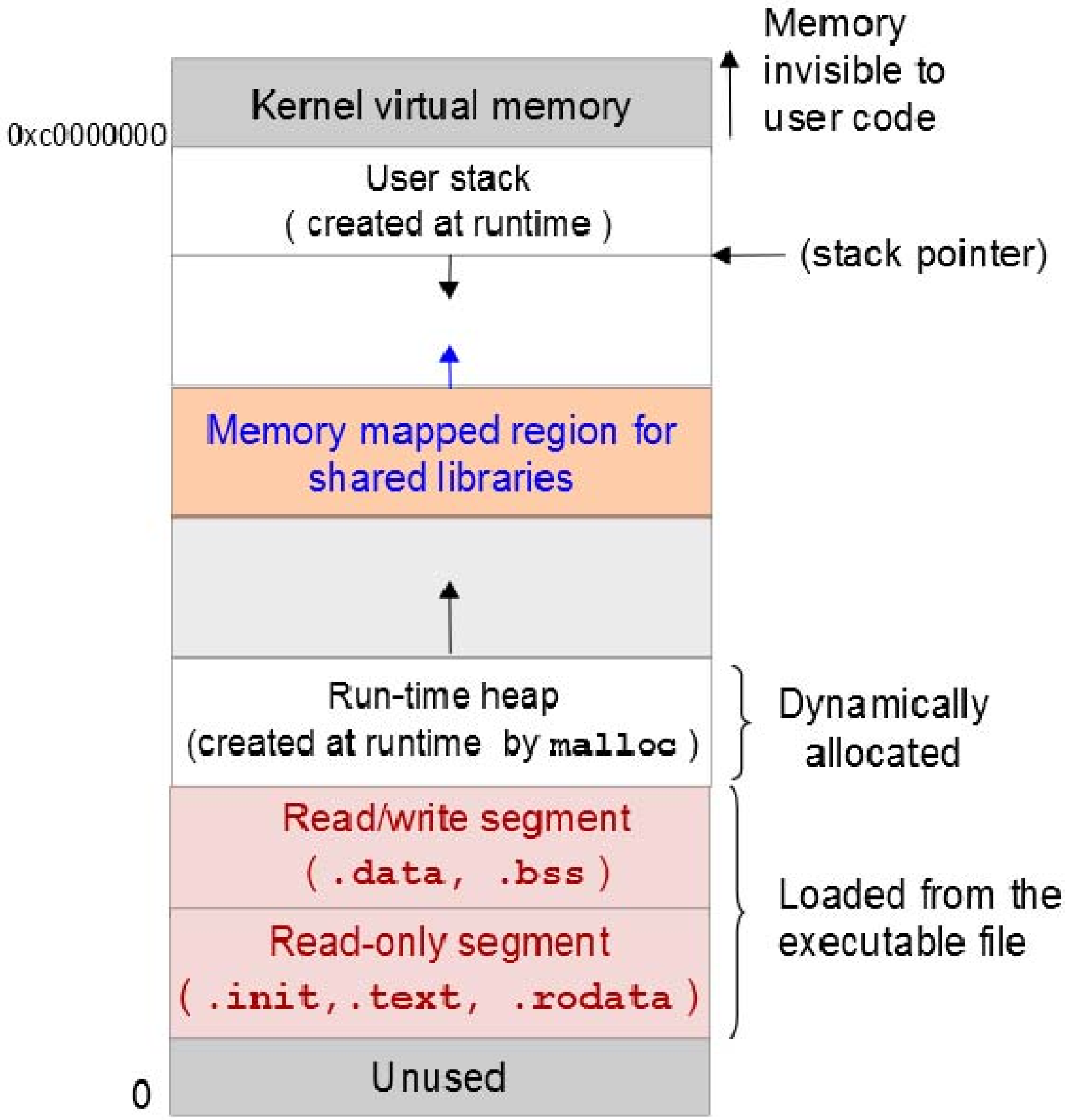
- There are ***three different*** (independent) ***measurements*** that quantify memory use of a program
  1. **Static** Memory (memory taken **by "code"**)
  2. **Dynamic** Memory (**heap** memory)
  3. Stack Memory
- In general ***measuring memory use is harder*** than measuring execution time (no easy way to instrument a program to measure dynamic and stack memory)

ELF - Executable and Linking Format

Object files (.o or .obj) are ELF files



Memory Use of (ELF) Program



Keep Track of:
1. Static Mem.
2. Dynamic Mem.
3. Stack Mem.

Reference: Computer Systems: A Programmer's Perspective
***Bryant* and O'Hallaron**

Performance: Static Memory Use

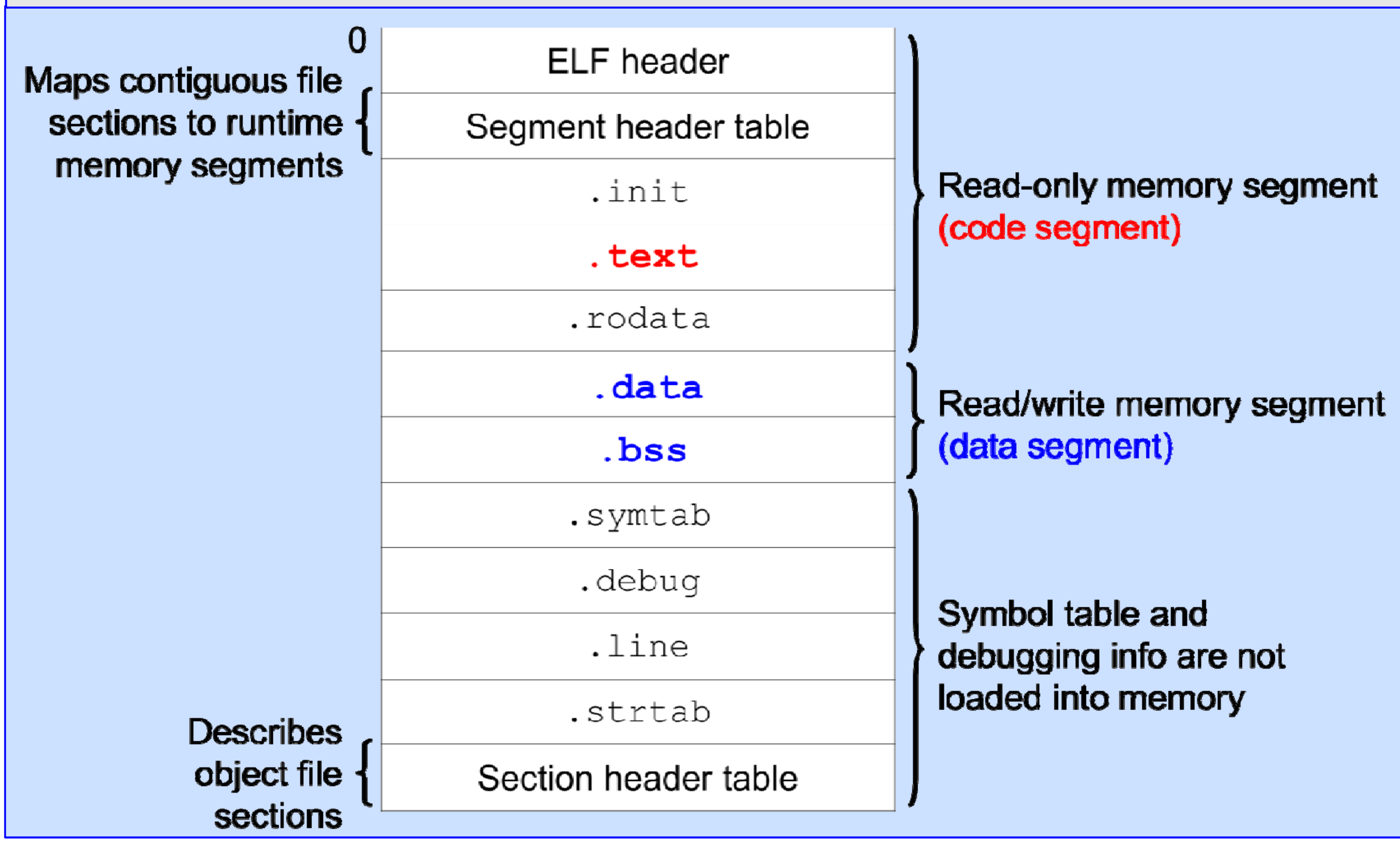
- In GNU/Linux ***static memory*** use can be "measured" using the `size` program.

- **size** reads an executable an reports the size of the following **ELF** ***program segments*** :
  - ***text segment*** **(`.text`):** contains all ***machine code***
  - ***data segment*** **(`.data`):** contains all statically initialized and global variables with nonzero values
  - ***bss segment*** **(`.bss`): b**lock **s**torage **s**tart, contains global and statically allocated data variables initialized to zero

Note: This description is for ELF files (Executable and Linking Format)

Size program

```
size reports memory footprint of:
   .text - your code segment
   .data - your non-zero data segment
   .bss  - your zero data segments
```



Performance: Static Memory Use

- The `.text` segment ***correlates well with memory footprint of the program*** (it tells how large the "code" is)

```
size of DisabledTiming  & EnbledTiming
   text     data      bss      dec      hex filename
   1057      264        8     1329      531 DisabledTiming
   2217      276       28     2521      9d9 EnabledTiming
```

- The choice of linking (static vs dynamic) also impacts the static memory footprint of the program

```
Compiled with -static-libgcc and -shared-libgcc
size shared & static
   text     data      bss      dec      hex filename
   2801      540       64     3405      d4d shared
   2741      524       64     3329      d01 static
```

Performance: Dynamic Memory Use

- Measuring dynamic memory use is very difficult without extensively instrumenting the program (see handout for details).

- In principle the ***heap manager*** (software that implements `malloc` and `free`) ***can report statistics of memory allocated*** at one time, average memory allocated and average size of blocks allocated

Reminder

heap usage with valgrind

http://valgrind.org/docs/manual/ms-manual.html#ms-manual.running-massif

- Compile code with **–g** option

- Invoque

```
valgrind --tool=massif ./TestDarray
```

Result: files with names **massif.out.####** (some number)

- Examine results of using **ms_print**

```
ms_print massif.out.23721 | more
```

- **ms_print** will produce:
  - A ***graph*** of program use of memory during execution
  - Details about allocation at various points in the execution

Partial valgrind example

The program dynamically reads 206,590 words into a 256 byte long string.
Expected memory usage = 50.4 MB

```
valgrind --tool=massif ./TestDarray
ms_print massif.out.23721 | more
```

```
    MB
51.23^                                                                       #
     |                                                                    :::#
     |                                                                :@:::::#
     |                                                             ::::@:::::#
     |                                                          :@:::::@:::::#
     |                                                      @::::@:::::@:::::#
     |                                                  ::::@::::@:::::@:::::#
     |                                               ::@:: :@::::@:::::@:::::#
     |                                           ::::::@:: :@::::@:::::@:::::#
     |                                        @::: ::::@:: :@::::@:::::@:::::#
     |                                    ::::@::: ::::@:: :@::::@:::::@:::::#
     |                                 ::@:: :@::: ::::@:: :@::::@:::::@:::::#
     |                             ::::: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |                         :::::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |                     ::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |                  :::::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |               :::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |          :@@@::::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |       ::@:@ @::::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |    :@@: @:@ @::::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
   0 +----------------------------------------------------------------------->
```

Performance: Stack Memory Use

- Measuring stack memory use is one of the most difficult tasks.
- ***Stack memory*** use in **embedded system** is ***critical*** since the **stack space** is usually **fixed** and pre-allocated.
- In machines with more resources (such as servers, etc.) the size of the ***stack can grow*** (limited only by the amount of memory available).

(In this course we will not measure stack memory use)

Code timing Problem 1

Will these macros properly measure code performance?
Why or why not?

```
START_TIMER(input);
BEGIN_REPEAT_TIMING(repeat_counter,10000);
 InputFile = fopen(argv[1], "r");
 do {
    if (fscanf(InputFile, "%lf %lf", &X, &Y) != EOF)
          {AddPoint(&DataSet, X, Y);  Done = 0; }
   else { Done = 1;   }
  } while (!Done);
END_REPEAT_TIMING;
STOP_TIMER(input);
fclose(InputFile);
```

Not a chance!
In pass 1 everything works as expected, BUT in pass 2 …N all the data has been read so NO MORE DATA IS READ giving a false measurement!
How do we fix this bug?

Code timing Problem 2

Will this properly measure code performance?

```
#include <stdio.h>
#include <time.h>
#define REPEAT 10000L
int main () {
  int       A, B, C, Index;
  clock_t  StartTime,  ElapsedTime;
  A=15; B=3;
  StartTime = clock();
     for (Index = 0; Index < REPEAT; Index++) { C=A*A/B; }
  ElapsedTime=  clock() - StartTime;
 printf ("   Total Time = %ld \"clock ticks\"\n", ElapsedTime);
 exit(0); }
```

Not a chance!
We are trying to measure a math operation with only 10,000 loops.
The answer is always ZERO!
How do we fix this bug?

REPEAT needs to be more like 10,000,000