CMPE-380: Applied Programming

Laboratory 06 Exercise

Macros & 2-D Arrays

Table of Contents

C Preprocessor, Macros, 2-D Arrays

Pre-Lab – 10 pts

In this lab session, we will review the C preprocessors, macros and dynamic 2D arrays. Review the class macro notes and read the following. Your TA will conduct a brief oral quiz.

The C Preprocessor

The C Preprocessor always runs before the compile process. The C preprocessor is a feature of C compilers but is not part of the C language. The C preprocessor is a string substitution language that inserts text into the source code. Often, the text inserted is C code but does not have to be.

Lines that begin with a **#** are preprocessing directives. The most common preprocessing directives are:

- **#include** used to include (header) files.
- **#define** used to define macros.

Include Directives

The **#include** preprocessor is used to include header files in a C program. Always include standard include files FIRST then personal include files. Personal include file names are included within double quotation marks, not <>. Never include .c files, it violates portability rules!

Example:

```
#include <stdio.h>          // Standard include file
#include <stdlib.h>
#include “myUtils.h”        // personal include file
```

Macros

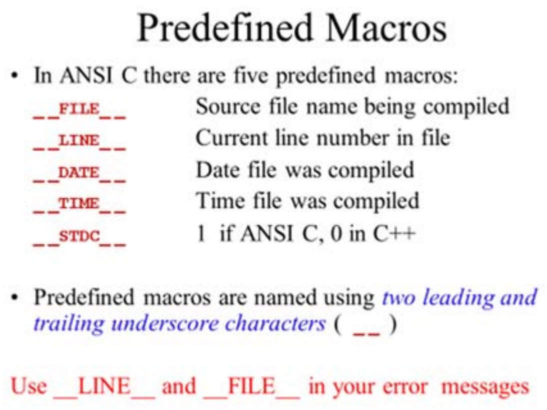
A macro is defined in C using the **#define** directive. A macro is a piece of string substitution code with a unique name. Macros are traditionally named with UPPER CASE and this simplifies debugging. Simple macros are used in C to declare "constants" which should be all CAPS. Example:

```
#define COUNT  (100)                          // note the parenthesis
```

- Macros can use parameters. Example:
  - #define SQ(x) ((x) * (x))
  - #define CUBE(x) (SQ(x) * (x))

  Note: parameters in a macro are just strings and so are typeless.

- Use **#undef** to remove a macro you don't want.
  - #undef SQ
- In ANSI C there are five very useful predefined macros shown in Figure 1.



Predefined Macros

- In ANSI C there are five predefined macros:

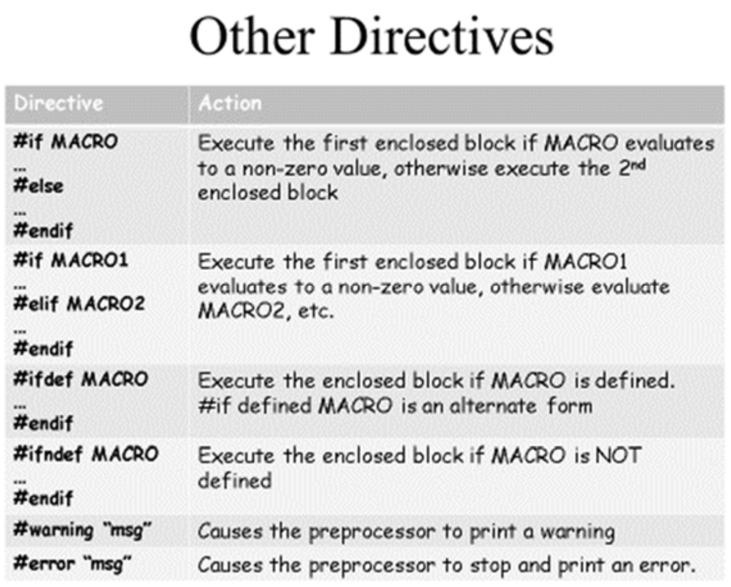
`__FILE__`	Source file name being compiled
`__LINE__`	Current line number in file
`__DATE__`	Date file was compiled
`__TIME__`	Time file was compiled
`__STDC__`	1 if ANSI C, 0 in C++

- Predefined macros are named using *two leading and trailing underscore characters* (__)

Use __LINE__ and __FILE__ in your error messages

Figure 1 Predefined macros

- Macro operators # and ##
  - # converts a macro parameter to a quoted string
  - ## concatenates
  - Examine the provided code **example/lab_b.c** Build and run it. What does it print?
- A list of additional macro directives are listed in Figure 2



Other Directives

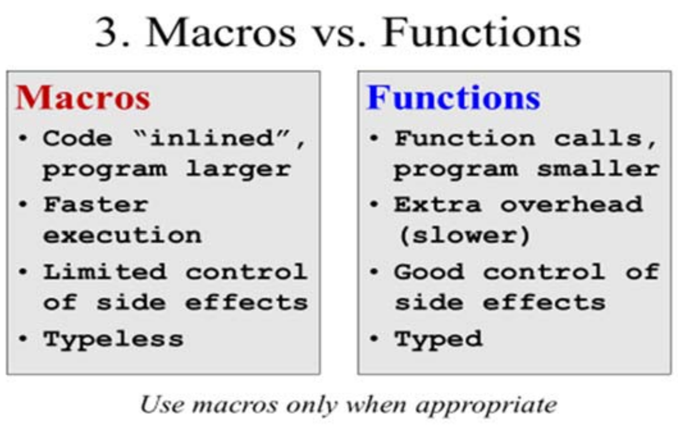
Directive	Action
#if MACRO ... #else ... #endif	Execute the first enclosed block if MACRO evaluates to a non-zero value, otherwise execute the 2nd enclosed block
#if MACRO1 ... #elif MACRO2 ... #endif	Execute the first enclosed block if MACRO1 evaluates to a non-zero value, otherwise evaluate MACRO2, etc.
#ifdef MACRO ... #endif	Execute the enclosed block if MACRO is defined. #if defined MACRO is an alternate form
#ifndef MACRO ... #endif	Execute the enclosed block if MACRO is NOT defined
#warning "msg"	Causes the preprocessor to print a warning
#error "msg"	Causes the preprocessor to stop and print an error.

Figure 2 Other directives

Macros vs Functions

As you see in Figure 3, there are significant differences between functions and macros. If you do not know that, you might struggle to maintain your code.

Follow a consistent convention (coding/naming) for Macros.



3. Macros vs. Functions

Macros	Functions
• Code "inlined", program larger	• Function calls, program smaller
• Faster execution	• Extra overhead (slower)
• Limited control of side effects	• Good control of side effects
• Typeless	• Typed

Use macros only when appropriate

Figure 3 Macros vs Functions

You should be prepared to write very simple macros and be able to explain the use of #ifdef, #ifndef and the special __FILE__ and __LINE__ macros. You should be able to explain what makes a macro different than a function. You should be able to explain the difference between the “#” and “##” macro metacommands. You should know how to use the line continue operator “\”.

2 Dimensional Arrays

In C, arrays of primitive types are implemented as blocks of contiguous memory.

```
int a[10];          /* 1D array of 10 integers */
```

In the case of a simple 1D array, the compiler "storage mapping function" will calculate the memory offset by taking the integer length (4) and multiplying by the index value.

e.g.   a[3]   is at memory location   &a + 4*3 (bytes)   // assumes an int is 4 bytes

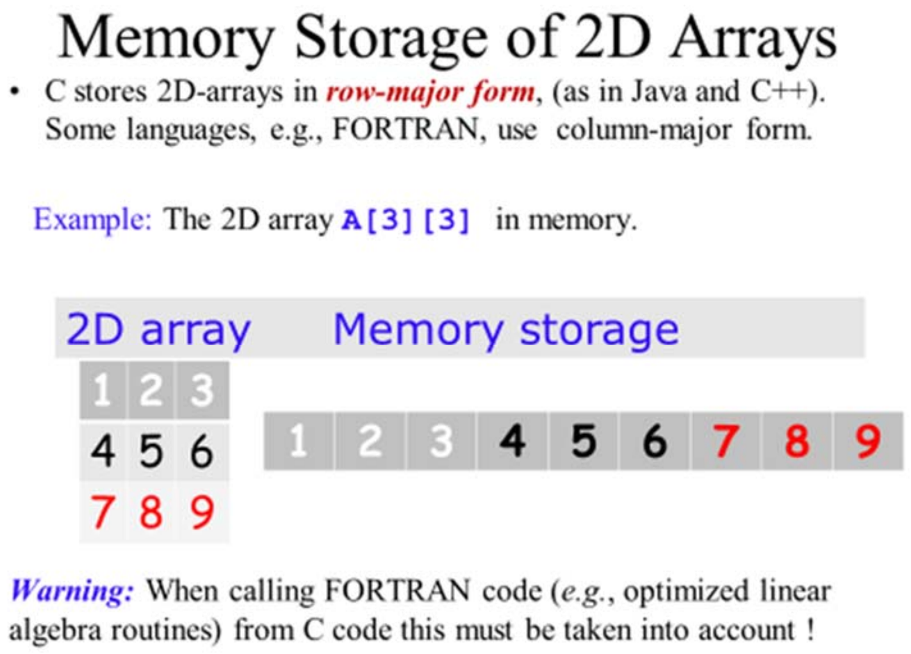
Notice that the storage mapping function will work for statically defined arrays and for malloc() memory pointers, all the compiler needs to know is the length of the data type.

e.g.   **int *a = malloc(10)**;   is functionally identical to   **int a[10]**

Note: The following NEVER works in C:

```
int x = 10            // x gets the value 10 at run time
int a[x];             // This results in ONE BYTE.  Memory is allocated at
                      // compile time NOT run time.
```

Two dimensional arrays are more difficult to access because the 2D object must be mapped into 1D physical ram. C (Java and C++) choose to use a “row major” format:



Memory Storage of 2D Arrays

- C stores 2D-arrays in ***row-major form***, (as in Java and C++). Some languages, e.g., FORTRAN, use column-major form.

Example: The 2D array **`A[3][3]`** in memory.

Warning: When calling FORTRAN code (*e.g.*, optimized linear algebra routines) from C code this must be taken into account !

As you can from this example the location of the row major index is dependent on the length of the row. The C “storage mapping function” is more complicated and must know the length of a row ahead of time..

Consider the following:

```
int A[3][5];            // 2D array of ints (“3 rows, 5 columns”)
```

To find the memory address of A[3][5] the compiler storage mapping feature must generate the following calculation:

e.g. A[2][3] is at memory location &A + (5*2+3)*4 (bytes) // assumes an int is 4 bytes

The compiler must know the data type AND THE LENGTH OF A ROW to calculate the offset. This means that a simple malloc() will not work to dynamically allocate 2D memory.

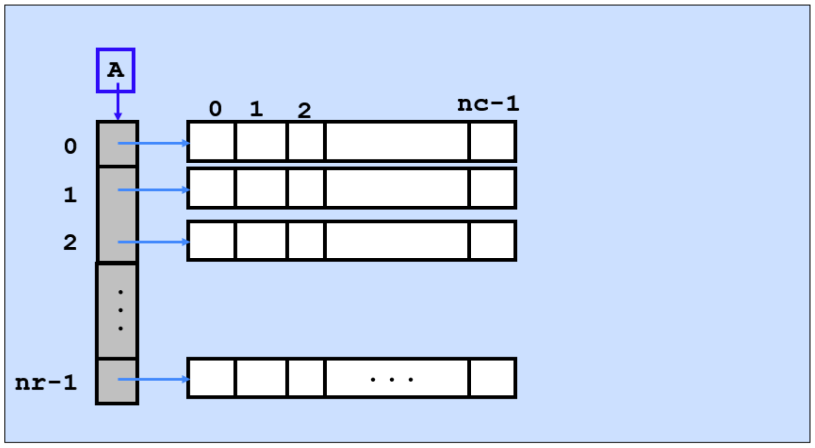
Note: The following NEVER works in C:

```
int x = 10          // x gets the value 10 at run time
int y = 20;
int a[x][y];        // This results in ONE BYTE.  Memory is allocated at
                    // compile time NOT run time.
```

To allocate a 2D dynamic array in C we need to be clever. We must use multiple 1D arrays to implement a 2D array in a way that looks like a real 2D array but is really two lookups of a 1D arrays. We are going to use some simple pointer trickery.

Consider the following design using a 1D "row" of pointers (A) to point to individual 1D "rows" of data.

Dynamically Allocated Arrays



Dynamically Allocated 2D Array in Memory ("row-major" form)

1) Define "A" to be a pointer to an array of pointers: e.g: **int **A**
2) Malloc "rows" rows in the 1D A array: e.g: **A = malloc(rows*sizeof(int *));**
3) Loop through the row pointer matrix allocating the actual row entries "cols"

   e.g **A[i] = malloc(cols* sizeof(int));** // A[i] is of type int* due to the ** definition.

When using the variable A[2][3] the compiler really decodes this as:

(A[2]) [3]. A[2]   // **(A[2])** returns a 1D pointer which is then used to access
// the 1D data array row. The [3] then indexes into that row.

Array Access Example

1) Read and understand the provide code lab_a.c
2) Build the provided lab_a.c code as lab_a (*gcc -g -std=c99 -o lab_a lab_a.c*) and then run it (*./lab_a*). Do all access methods print the same value?

You should be prepared to explain to the TA why trying to create dynamic arrays using variable in the index won't work and how this 2D approach does work. You should be able to calculate the storage mapping function for simple 1D and 2D arrays. You should be able to explain how each of the storage accessing methods in the program works.

Interactive Exercises – 30 pts

Redirect the output from each of the steps as directed into **exercise.txt** and show the results to your TA when you are finished. You will be documenting your comments in exercise.txt, be sure to clearly indicate what step you are commenting on. You will want to test each stage of your exercise redirecting the output into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

Macros

This exercise illustrates the power of macro commands.

1) Compile the provided code lab_b.c as lab_b1 with “-DTEST=1” macro.

```
gcc  -g  -Wall  -std=c99 -pedantic  lab_b.c  -o lab_b1  -DTEST=1
```

2) Run: ./lab_b1 > exercise.txt

3) Compile: lab_b.c using the -E -P macro expansion flags, remove the -o lab_b1 option and append the output to exercise.txt. The -E -P flags will cause the compiler to dump the macro preprocessor.

```
gcc  -g  - Wall  -std=c99 -pedantic  lab_b.c   -P  -E    -DTEST=1  >>  exercise.txt
```

4) Edit exercise.txt and identify the expanded macro code (hint: look for “main”) and then document the original C macro calls to the actual macro expansion in the **exercise.txt** file as comments. (e.g copy the original C code fragment and the resulting macro expansion fragment) Notice the variable names in the for loop. The purpose of this step is to show that macros just result in text which the compiler can treat as code.

5) Explain what the “#if TEST==1” macro does in your analysis.txt file.

6) Try to re-compile the lab_b.c as lab_b2 with the “-DTEST=2” macro enabled on the command line, and no -E -P, it will not compile. Document the error in exercise.txt.

```
gcc  -g  - Wall  -std=c99 -pedantic  lab_b.c  -o lab_b2  -DTEST=2
```

7) Re-compile lab_b.c -DTEST=2 with the -E -P expansion flags, and no -o. Identify the exact offending code. How did the “bad” code get in there? Include your macro expansion in your **exercise.txt** file.

```
gcc -g -Wall -std=c99 -pedantic lab_b.c -E -P   -DTEST=2 >> exercise.txt
```

8) Fix the offending macro by prepending an underscore “_” to the macro index variable and loop code as necessary and recompile. Do not change any other variable names.
9) Re-run the ./lab_b2 code and append the output to your exercise.txt file.

10) Re-compile the lab_b.c as lab_b3 with the “-DTEST=3” macro enabled on the command line and run it.
11) Does it print 50? If not, why? Support your conclusion with proof from the macro expansion in exercise.txt.

12) Re-compile the lab_b.c as lab_b4 with the “-DTEST=4” macro enabled on the command line and run it.
13) Is the output what you expected? Support your conclusion with proof from the macro expansion in exercise.txt.

14) Re-compile the lab_b.c as lab_b5 with the “-DTEST=5” macro enabled on the command line and run it.
15) How many charCount() function call seem to be called in the code for test 5? Comment in your exercise.txt.
16) How many charCount() function calls really made in the code? Comment in your exercise.txt. Support your conclusion with proof from the macro expansion in exercise.txt.

2-D Arrays

This exercise illustrates a simple 2D dynamic macro array and the value of Valgrind to find hidden issues.

1) Read and understand the lab_c.c code.
2) Compile the lab_c.c code and run it (*gcc -g -std=c99 -o lab_c lab_c.c*). Does the code do what you expect? Are there any apparent bugs? Comment in your exercise.txt code.

3) Run Valgrind on lab_c. (valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./lab_c). Are there any bugs? What are they? Include your Valgrind output and comments in your exercise.txt code.

4) Implement a simple "free(A)" and recompile.
5) Run Valgrind again. Did this "fix" the problems? Did this "improve" the problems? Include your Valgrind output and comments in your exercise.txt code.

6) Comment out the simple free() function
7) Write the free_matrix_space() subroutine and call it from main.
8) Run Valgrind again. Did this "fix" the problems? Did this "improve" the problems? Why do we need to include "rows" in the subroutine. Include your Valgrind output and comments in your exercise.txt code.

9) The "A" matrix is a "dynamic matrix", write code in the indicated section to add 5 rows to the existing "A" matrix. The code to add 5 rows must NOT delete, copy or modify the previous data. You must update the "A" row pointer variable and add new rows using malloc/calloc/realloc as you see fit. Your new additional row data should be initialized as before but with a negative value. The number of columns will remain unchanged.

10) Add code to print out the new 8x5 matrix in the identified code area. The resulting program output should look like:

```
The 3x5 2D dynamic matrix
 0.0   0.1   0.2   0.3   0.4
 1.0   1.1   1.2   1.3   1.4
 2.0   2.1   2.2   2.3   2.4
The 8x5 2D dynamic matrix
 0.0   0.1   0.2   0.3   0.4
 1.0   1.1   1.2   1.3   1.4
 2.0   2.1   2.2   2.3   2.4
-3.0  -2.9  -2.8  -2.7  -2.6
-4.0  -3.9  -3.8  -3.7  -3.6
-5.0  -4.9  -4.8  -4.7  -4.6
-6.0  -5.9  -5.8  -5.7  -5.6
-7.0  -6.9  -6.8  -6.7  -6.6
```

11) Rerun valgrind and fix any memory leak issues you might have. Include your Valgrind output in your exercise.txt code.

Assignment – 60 pts

Objectives

To develop a reusable module with timing macros for instrumentation of C programs.

Setup

Upload the file **hw6_files.tar,** available in MyCourses, to a suitable directory in your cluster account. Then unarchive it (**tar -xvf hw6_files.tar**).

This will create the directory **hw6** with files: **hw6cpp.cpp, hw6c.c, sleep.c, data.txt and Timers.h**.

Note that **Timers.h** contains partial timing macros, you will write your implementation of timing macros here.

The files **hw6cpp.cpp** and **hw6.c** implement a least-squares linear fitting program, in C and C++ . These programs take their *input from a file* whose name is passed as a *command-line argument* and are designed to accommodate a *variable number of data points*. The file **data.txt** contains the data points to be used as input to the programs *for testing*. Both programs read all the data points from the input data file and store them in memory for "batch processing" (The number of data points is not known in advance). The C program uses dynamic arrays to store the data. **sleep.c** implements a simple 60 second delay program which can be used to test your timing macros.

Preliminary Step

Compile the C and C++ programs and examine the output and the code to understand what they do and how they do it (note the use of dynamic arrays in the C program). Use the following directives for compilation:

a) C++ compilation command: **g++ -Wall -pedantic -std=c++98 -O1 hw6cpp.cpp -o hw6cpp**
b) C compilation command: **gcc -Wall -std=c99 -pedantic -O1 hw6c.c -o hw6c**

Note: **g++** and **gcc** are DIFFERENT compilers.

Implementation of Instrumentation Macros

Implement the **Timers.h** module as described in Chapter 2, Listing 2.1 of Greg Semeraro's Book – (posted in MyCourses **[References]**). You ***must use these macros*** to instrument the programs in this homework and asses their performance. Note that you will not receive any credit if you "hard code" timing statements without using **Timers.h**. In Listing 2.2 of the reference you will find an example that describes how to use these macros.

In addition to the macros described in the given reference you will implement three additional macros:

1. **PRINT_RTIMER(A,R)** This macro takes a timer name **A** and a number **R** representing the number of repetitions (of the code being timed) to print the correct timing info, that is, the time reported by **PRINT_TIMER(A)** divided by **R**
2. The macros **BEGIN_REPEAT_TIMING(R, V)**, **END_REPEAT_TIMING** should be designed to wrap the code begin timed in a for–loop that executes the code **R** times. You will need another macro, **DECLARE_REPEAT_VAR(V)** to allocate a looping variable **V**. Remember that macros are really just text so they can include fragments of "C" code.

These macros should behave in the same way that all the other **Timer.h** macros do, that is, they will be included in the code when **EN_TIME** is set and expand to "nothing" otherwise. Be sure to verify the "nothing" case.

Instrumentation for Timing: For timing of the C and C++ program take into consideration the following *requirements*:

1. Your timing should *measure and report execution time* of ***data input*** and ***calculations*** separately. Use two different timers, one for data input and another for calculations. Your data should include the total elapsed time and the per loop time for data input and calculations.
2. You will need to utilize the "repeat timing" macro features to achieve accurate results.
3. Your C program timing should be "fair" compared to the C++ program, that is, the C timing should include, as much as possible, the same processing as the C++ timing.

   Run your input data measurement code three times without modification. You will find that data input performance is HIGHLY variable. This variability significantly impacts our ability to "tune" our software because we will have issues identifying the value our changes versus normal system performance noise.

   Both the C and CPP code do file IO. As we covered in class, it is often necessary to restructure code to make it measurable. A macro variable called "**MOVE_FOPEN**" was added to the C code to "move" the IO open operation from inside the timing loop to outside the timing loop. Measure the C performance with the IO open code in each position, noting the differences in performance. Which is "faster"? Is it "real"? Are the answers correct?

Code Instrumentation

The C and C++ files have clear comments included to help you place your macros in the proper location.

a) Instrument the C++ program using your **Timers.h** macros.
   For timing you will compile the C++ program as follows:

   g++ -Wall -pedantic -O1 -DEN_TIME hw6.cpp -o hw6cpp

b) Instrument the C program using your **Timers.h** macros.
   For timing you will compile the C program as follows:
   gcc -Wall -std=c99 -pedantic -O1 -DEN_TIME hw6.c -o hw6c

Testing Matrix

Each item in the matrix must be run 3 times and the final results reported must be the average of the runs. These test cases can be built and run by hand, they are not required to be in the make file. All resulting raw data must be clearly identified and provided.

Test	Item	Analysis
1	C code with NO MOVE_FOPEN option	
2	C code with NO MOVE_FOPEN option AND 2X the loop count of #1	Compare the results from #1 & #2. What happens to the per loop CPU and DATA performance? What happens to the total time for the CPU and DATA?
3	C code with MOVE_FOPEN option	
4	C code with MOVE_FOPEN option AND 2X the loop count of #3	Compare the results from #3 & #4. What happens to the per loop CPU and DATA performance? What happens to the total time for the CPU and DATA?
5	C++ code	Compare the results of #1, the C code, with #5 the C++ code. Can you draw any conclusions about the speed of IO and calculations between C & C++?
6	C++ code with 2X the loop count of #5	Compare the results from #5 & #6. What happens to the per loop CPU and DATA performance? What happens to the total time for the CPU and DATA?

Note: **matrix.txt** contains an example of the expected data. Due to implementation differences your performance numbers will be different. Some calculations will produce NAN (not a number), this is a side effect of the timing changes we are making and prove that our timing changes “broke the code”. You don’t have to correct this.

Makefile

You must include a quality makefile with: all, test, help and clean targets. You must use a macro variable to control the timing macros. Your code must compile and run cleanly with timing enabled or disabled. The makefile should enable the MOVE_FOPEN and EN_TIME macros by default.

1) "all" should build all your timing enabled code (hw6c and hw6_cpp).
2) "test" should run **hw6c & hw6cpp**, each, with **data.txt** and redirect all the output to a single file called "out.txt". A line indicating what command is being executed must be included in "out.txt" prior to the actual execution of each command.
3) "help" and "clean" do the normal things.

Once you are done create a **lastName_hw6.tar** (lastName is your last name) file and submit it.

Grading Criteria

a (12 pts) **Timers.h** module properly implemented with correct implementation of **PRINT_RTIMER(A,R)**, **BEGIN_REPEAT_TIMING(R,V), END_REPEAT_TIMING** and **DECLARE_REPEAT_VAR(V)**
b (8 pts) C++ program properly instrumented using **Timers.h**
c (8 pts) C program properly instrumented using **Timers.h**
d (14 pts) Make file is complete and of high quality
e (18 pts) Analysis of the data, including all raw data runs, averaged results and comparison questions addressed.

Hints

- You should verify your timing macros with the sleep.c code. If you later get a "zero" for the execution time, you know the problem must be in the number of loops in your repeat values, not a problem with the macros.
- Use the **-E** and **-P** compiler flags to SEE your expanded macros in the source code.
- You should expect IO operations to be at the rate of milliseconds and computations at the rate of 100's of nanoseconds.

Student Name: ____

Laboratory Grading Sheet

Lab 06 - Macros & 2-D Arrays

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: C Preprocessor, Macros	**5**		
Pre-Lab: 2D Arrays	**5**		
Interactive Exercises: Macros	**20**		
Interactive Exercises: Dynamic Arrays	**10**		
Total	**40**		

You must turn this signed sheet in at the end of lab to receive credit!