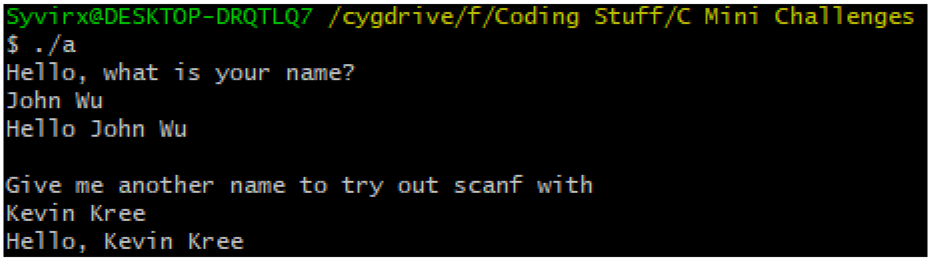
Question 1 – “Hello <NAME>”:

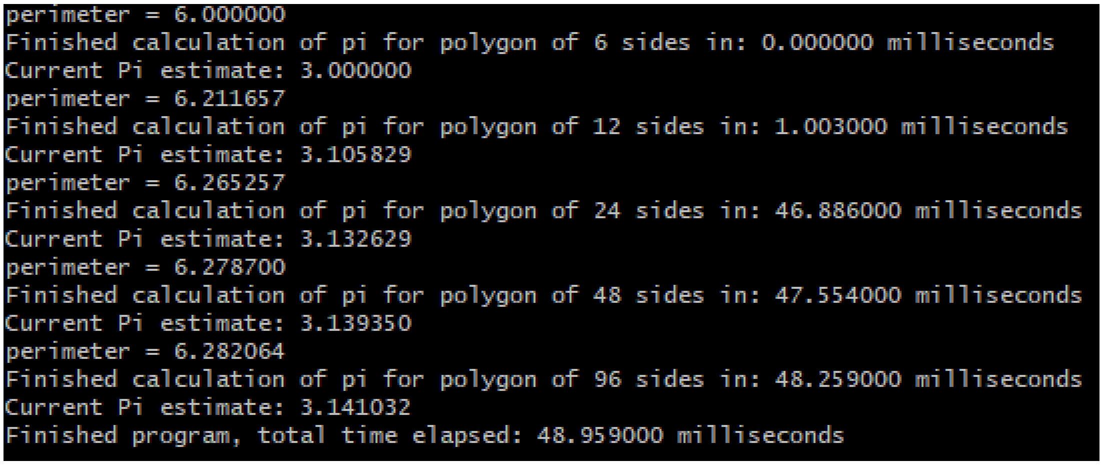


```
Syvirx@DESKTOP-DRQTLQ7 /cygdrive/f/Coding Stuff/C Mini Challenges
$ ./a
Hello, what is your name?
John Wu
Hello John Wu

Give me another name to try out scanf with
Kevin Kree
Hello, Kevin Kree
```

a. I actually used fgets(), because I saw that it had no buffer overflow protection on stack overflow. I’m quite forgetful like that. I found that with scanf() that the entry would end when you put a space. I got rid of this by having an extra field to accept more string entries, but more elegantly you could accept space characters or go until a new line (\n) is detected. This is exactly what I did with scanf(), I made my scanf continue until it reached the new line character.

Question 2 – Archimedes Algorithm

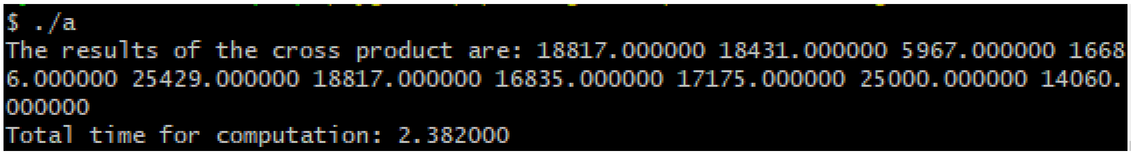


```
perimeter = 6.000000
Finished calculation of pi for polygon of 6 sides in: 0.000000 milliseconds
Current Pi estimate: 3.000000
perimeter = 6.211657
Finished calculation of pi for polygon of 12 sides in: 1.003000 milliseconds
Current Pi estimate: 3.105829
perimeter = 6.265257
Finished calculation of pi for polygon of 24 sides in: 46.886000 milliseconds
Current Pi estimate: 3.132629
perimeter = 6.278700
Finished calculation of pi for polygon of 48 sides in: 47.554000 milliseconds
Current Pi estimate: 3.139350
perimeter = 6.282064
Finished calculation of pi for polygon of 96 sides in: 48.259000 milliseconds
Current Pi estimate: 3.141032
Finished program, total time elapsed: 48.959000 milliseconds
```

a. I timed my program using the struct timeval and the function call gettimeofday() to get the time in micro seconds, then I divided the result by 1000 to get it in milliseconds.
b. I had some issues with precision for a while, but after rereading the slides, I realized that I was dividing by integers and not floats, which made all the difference.

*PS: I wasn’t sure if I was supposed to do both circumscribed and inscribed, so I tried doing the inscribed circle as well, but I couldn’t quite figure it out.

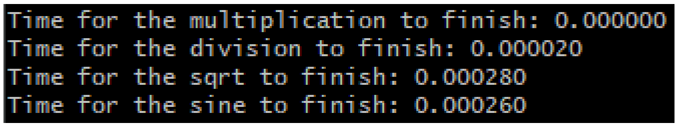
Question 3 – Mat X Vec:



```
$ ./a
The results of the cross product are: 18817.000000 18431.000000 5967.000000 1668
6.000000 25429.000000 18817.000000 16835.000000 17175.000000 25000.000000 14060.
000000
Total time for computation: 2.382000
```

a. I allocated and accessed my Matrix by making a matrix variable that was a single array, and I placed the subsequent values as a pointer to another array.
b. I had no challenges reading my file, I used the fgets and sscanf functions to do most of my file reading operations. I also used strtol to a limited capacity.
c. There was nothing really special about the actual computation, it just worked the same way matrix multiplication always has.
d. I just timed it with the sys\time.h gettimeofday and timeval struct. I did not check for dynamic memory succession, I would've added a check to make sure it didn't return null if I did, however. I only checked to make sure the two could by multiplied together, I did not make sure that the vector was a 1xn array.

Question 4 - Speed of Computations:

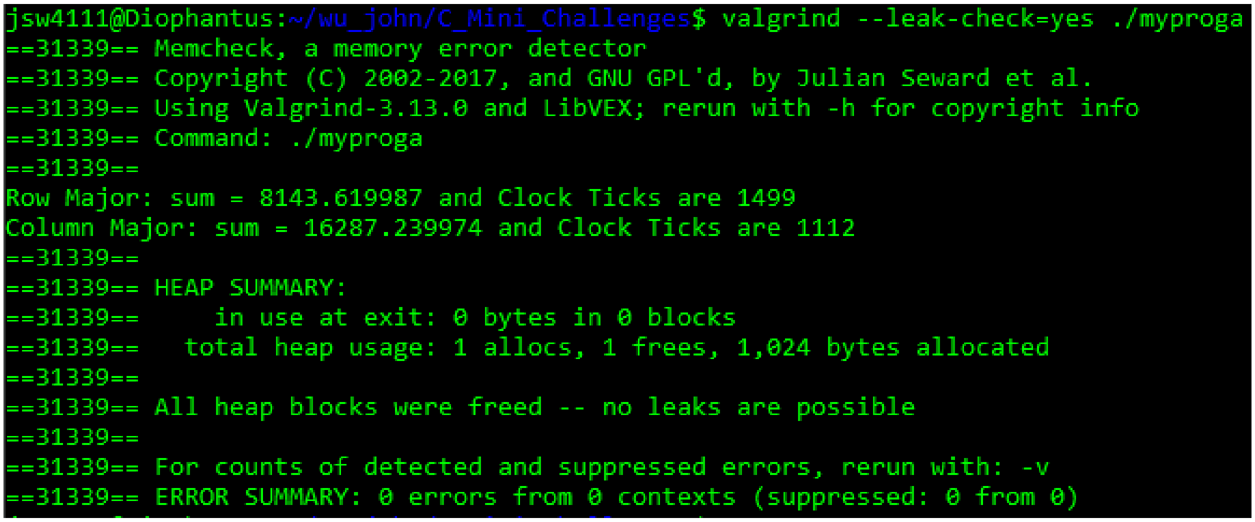


```
Time for the multiplication to finish: 0.000000
Time for the division to finish: 0.000020
Time for the sqrt to finish: 0.000280
Time for the sine to finish: 0.000260
```

a. My timing strategy was to compute 50 runs of each computation and take the average using the timeval struct and gettimeofday function.

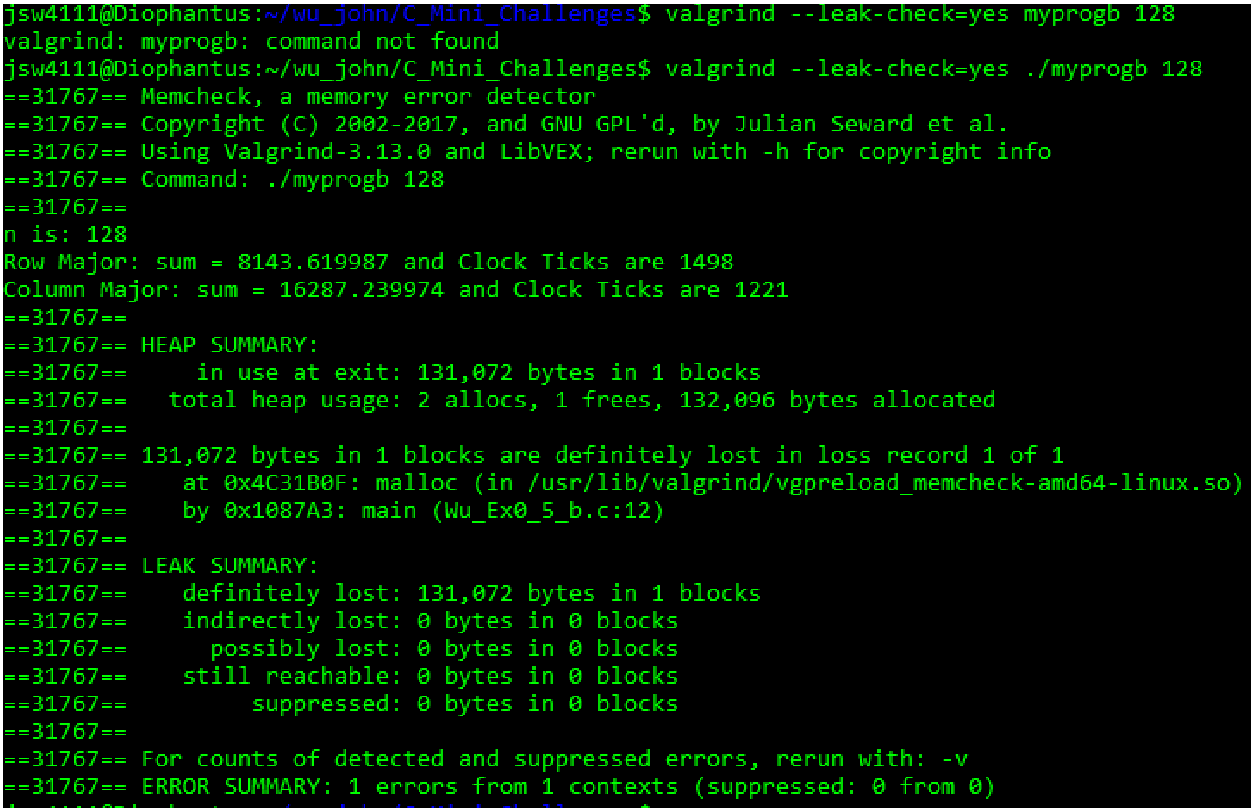
b. No, multiplication is faster than division, which is faster than sqrt(), which is faster than sin() on average. I think the differences lie with how the computer actually computes these functions, and my assumption that each process uses multiple uses of the ones that precede it.

Question 5 Part A - Statically Allocated Array:



```
jsw4111@Diophantus:~/wu_john/C_Mini_Challenges$ valgrind --leak-check=yes ./myproga
==31339== Memcheck, a memory error detector
==31339== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31339== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==31339== Command: ./myproga
==31339==
Row Major: sum = 8143.619987 and Clock Ticks are 1499
Column Major: sum = 16287.239974 and Clock Ticks are 1112
==31339==
==31339== HEAP SUMMARY:
==31339==     in use at exit: 0 bytes in 0 blocks
==31339==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==31339==
==31339== All heap blocks were freed -- no leaks are possible
==31339==
==31339== For counts of detected and suppressed errors, rerun with: -v
==31339== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Question 5 Part B - Dynamically Allocated Array:



```
jsw4111@Diophantus:~/wu_john/C_Mini_Challenges$ valgrind --leak-check=yes myprogb 128
valgrind: myprogb: command not found
jsw4111@Diophantus:~/wu_john/C_Mini_Challenges$ valgrind --leak-check=yes ./myprogb 128
==31767== Memcheck, a memory error detector
==31767== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31767== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==31767== Command: ./myprogb 128
==31767==
n is: 128
Row Major: sum = 8143.619987 and Clock Ticks are 1498
Column Major: sum = 16287.239974 and Clock Ticks are 1221
==31767==
==31767== HEAP SUMMARY:
==31767==     in use at exit: 131,072 bytes in 1 blocks
==31767==   total heap usage: 2 allocs, 1 frees, 132,096 bytes allocated
==31767==
==31767== 131,072 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31767==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==31767==    by 0x1087A3: main (Wu_Ex0_5_b.c:12)
==31767==
==31767== LEAK SUMMARY:
==31767==    definitely lost: 131,072 bytes in 1 blocks
==31767==    indirectly lost: 0 bytes in 0 blocks
==31767==      possibly lost: 0 bytes in 0 blocks
==31767==    still reachable: 0 bytes in 0 blocks
==31767==         suppressed: 0 bytes in 0 blocks
==31767==
==31767== For counts of detected and suppressed errors, rerun with: -v
==31767== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

a. The differences are that dynamic requires you to free the variable.
b. Between column major and row major, depending on your allocation scheme, the second major is more time inefficient. I think this is because when you follow the first major, you just read line by line, but with the second major, you kind of jump around, pointing from a memory cell ahead of your next one, needing to jump back.