HELP SESSION 1
HW1 and C Review

CS370
COLORADO STATE UNIVERSITY
BY TOMAS VASQUEZ
Outline

• Overview of the assignment
• Pointers and references
• Dynamic Memory
• Tying it all together/Questions
Overview of Assignment

- Required files:
  - Initiator.c
  - Worker.c
  - Worker.h
Initiator.c

• Takes in one command line argument
  ⇒ Perform argument check
• Set the seed with srand()
  • atoi()
• Invoke functions in worker.c
  • float running_ratio = get_running_ratio();
• What should be included in initiator.c so that it can call the functions in worker.c?
Worker.c

• int random_in_range(int lower_bound, int upper_bound)
• float get_running_ratio();
• int get_divisibility_count (int *array, int arraySize, int randomDividend);

You are encouraged to define new functions as you see fit. However, the above three functions must be included.
int random_in_range(int lower_bound, int upper_bound)

• Returns a random number in range \([a,b)\)
• Given to you in write up
int get_divisibility_count (int *array, int arraySize, int randomDividend);

• Returns to get_running_ratio() the number of divisible items in each array.
float get_running_ratio();

• Controls flow of the program

1. Calculate the number of iterations for your loop
2. Allocate an array with the appropriate number of elements on each iteration
3. Generate a new divisor per iteration
4. Calls get_divisibility_count()
5. Keep track of the iteration with the largest number of divisible integers.
6. Keep a running sum of the ratio of (divisible/non-divisible)
7. Returns average ratio across all iterations
   => (running sum-from step (6))/number of iterations-from step (1)
C review

• The following slides are based on material gathered from CS370- Spring2021 Help Session 1.

• Materials and images found on the following websites:
C review: Pointers

• A pointer declared `datatype *var_name` is a reference to a section of memory allocated for some type of object.

• `*` operator is the de-referencing operator.
  
  It has dual meaning.
  
  1) declaring a pointer  
     ```c
     int *p;
     ```
  
  2) Accessing what the pointer is pointing to 
     ```c
     printf("%d", *p);
     ```

Warning regarding dangling pointers!!!!
References

- The & operator is used to obtain the address of an object so that it may be assigned to a pointer.
- Let int *p;
- If int x = 5; and p = &x
- Then return *p equals?

Image taken from:
https://iq.opengenus.org/pointers-in-c/
References continued

• Use the & operator to pass an object by address.
• Why?
• Its less costly than copying the object.

```c
void passByValue(int n) {
    a = 5;
}

void passByAddress(int *b) {
    *b = 10;
}

int main(void) {
    int c = 10;
    passByValue(c);
    passByAddress(&c);
    return 0;
}
```

Image taken from:
https://iq.opengenus.org/pointers-in-c/
Arrays

• Declaring an array

    Data_type array_name [ array_size ];

    Data_type array_name[n] = {x0,x1,x2,x3, ... xn-1} where (x0,..,xn-1) are objects of the data_type and n is the size of the array.

    NOTE: this is how you declare and initialize an array on the stack. Your assignment requires you to do so on the heap. More on that next!

    NOTE: [n] may be omitted in favor of []. Which implies you do not have to give a size when you declare and initialize in the same step.

• Indexing in arrays –zero based index

    Array_name[0] = 5

    return Array_name[0] -> returns 5
Arrays as pointers

- `Int my_array[] = {1,2,3,4,5};`
- `Int *p = my_array;`
- What does `p` contain? What about `*p`?
- `Int x = *(p+1)` equivalent to `x = p[1]`
- `p = &my_array[2]`
- What does `p` contain?
- `My_array[i]` is equivalent to `*(my_array+i)`
More operations on pointers

• *p++
  Says give me the value at p, then increment p such that it points to the next element.
  By how much is it incremented?

• *++p
  Says increment p and give me the value that p is now pointing to.

• ++*p
  Says increment the value at p
Pointers and Strings

- A string in C is an array of char types.
- It is terminated by ‘\0’ which is the null character.
- char my_string[] = “Hello World!”
- What is the size of my_string?
- Check it yourself
  - printf("%lu\n", (sizeof(my_string)/sizeof(char)));
Arrays as pointers

```c
int array[10];
int *ptr1 = array;
ptr1[0] = 1;
*(array + 1) = 2;
*(1 + array) = 2;
array[2] = 4;
```
THE HEAP!

- Your assignment requires you to allocate on the heap.

  - `void* malloc(size_t size);`
    - "allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block."
    - `malloc()` doesn’t initialize the allocated memory.

  - `void* calloc(size_t num, size_t size);`
    - Similar to `malloc` but initialises the memory to zero.

https://www.geeksforgeeks.org/difference-between-malloc-and-calloc-with-examples/
Free() and Valgrind

• You need to free the memory you allocate

• How do you check for memory leaks?
  Valgrind: A program for tracking memory leaks and errors.
  Command: valgrind -q --leak-check=full ./a.out
  To see the line where the memory leak occurred compile with –g flag
  gcc -o test -g test.c
  See this helpful lab from CS253 taught by Jack Applin for instructions on how to use valgrind
  NOTE: This lab is in c++ but valgrind works all the same.
  Please don’t use c++ code or compiler.
  https://www.cs.colostate.edu/~cs253/Fall21/Lab/Valgrind
Sample array on the heap and using free()

```c
// C program to demonstrate the use of malloc() and calloc()
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // malloc() allocate the memory for 5 integers
    // containing garbage values
    int* arr = (int*)malloc(5 * sizeof(int)); // 5*4bytes = 20 bytes

    // Deallocates memory previously allocated by malloc() function
    free(arr);

    // calloc() allocate the memory for 5 integers and
    // set 0 to all of them
    arr = (int*)calloc(5, sizeof(int));

    // Deallocates memory previously allocated by calloc() function
    free(arr);

    return (0);
}
```

Image taken from:
https://www.geeksforgeeks.org/difference-between-malloc-and-calloc-with-examples/
Makefile

• A Makefile is simply a way of associating short names, called targets, with a series of commands to execute when the action is requested
  • Default target: make
  • Alternate target: make clean
Makefile continued

- Basic macro: CC=gcc
  - Convert a macro to its value in a target: $(CC)
  - Ex: $(CC) a_source_file.c gets expanded to gcc a_source_file.c
- To execute: make / make build • To clean: make clean
Makefile Sample

- files=Program1.c Program2.c
- out_exe= Program1
- build: $(out_exe)
- $(out_exe): $(files)
  $(CC) -o $(out_exe) $(files)
- package:
  tar -cvzf John_Doe.tar *.c *.h *.txt Makefile
- clean:
  rm -f $(out_exe)
Any questions?
Acknowledgements

• These slides are based on contributions of current and past CS370 instructors and TAs, including Jack Applin, Abhishek Yeluri, Kevin Bruhwiler, Yashwant Malaiya and Shrideep Pallickara.