MPCS 51100 - Fall 2018
PSet 1
C Programming Review

Due Date: October 8, 2018 @ 5:30pm on Canvas

Introduction

- In this assignment, you will be implementing a variety of short programs in C.
- You should begin by cloning the github repository at `https://github.com/jtramm/51100_pset1`. The repository contains an API and functional prototypes that you will be filling in, as well as a testing framework. When you submit your code, you must maintain the same directory structure and filenames (e.g., p1.c, p2.c) so that your code can be compiled and run by an auto-grader automatically.
- In your top directory, you must populate the README.txt file to list your name, the assignment, and a discussion of any shortcomings your code may have. You will receive more partial credit if you are able to clearly identify faults or problems with your code, rather than letting us find them ourselves.
- Your code should be able to be compiled with the included makefile. That is, using gcc and with the following flags: `gcc -Wall -ansi -pedantic`. Notably, using this configuration will not allow support for C99 or C11 language features.
- Many of the problems in this assignment will be auto-tested by compiling your code and passing in a set of data via `stdin` or by file and comparing your code's output against a reference solution. This means that small deviations in formatting will result in the auto-grader reporting issues with your code. The grader will check any deductions to ensure they are substantial, but we would greatly appreciate you sticking exactly to the formatting requirements prescribed by the assignment. While you will not have deductions due to small formatting issues arising from ambiguous requirements, a total lack of effort on the part of the student to adhere to clearly stated formatting requirements (defined by given examples in this document and in given test code) of the assignment will result in deductions.
- If you are unable to implement the program correctly or fully, you may still receive partial credit, so turn in what you have done, and specify in the README where things may be problematic. If you have questions regarding expectations on the homework, please post your questions to Slack.
- Due date and extension policies are as prescribed by the course syllabus.
- The academic integrity policy for this assignment is as described in the course syllabus.

1 Heap - p1.c (10 points)

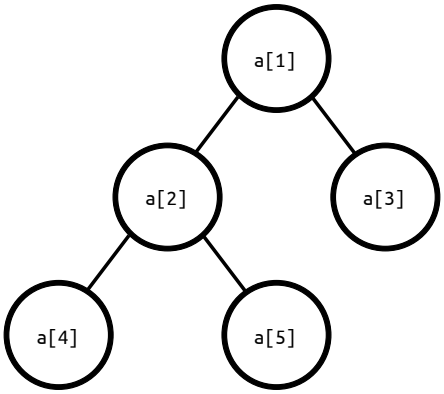
A "heap" is defined as a one-dimensional array, with indices from 1 to N, with the following property:

```
a[j/2] >= a[j]   for  1 <= j/2 < j <= N
```

where `N/2` means "integer divide" – that is, the result is truncated to the nearest integer. So for example:

```
a[1] >= a[2]; a[1] >= a[3];
a[2] >= a[4]; a[2] >= a[5];
```

We can visualize the heap as a tree with a[1] at the root:



Each "parent" must be greater than or equal to its two "children". In any case, this data structure turns out to be a very convenient arrangement for performing further work very efficiently, such as sorting the array.

Your task for this problem is to implement a `heapify()` function that arranges an arbitrary array of integers (taken as command line arguments) into a heap. A test main and function prototypes are provided for you already in p1.c – you should implement your code in this file following the prototypes provided. In addition to the `heapify()` function, you will also need to implement an `allocate_array()` function that takes the integers as input from `argv` and copies them to a separate dynamically allocated array after converting with `atoi()`. That is, you will need to implement the following two functions in p1.c:

```
void heapify(int *f, int n);
int * allocate_array( int argc, char * argv[]);
```

An example run should look like:

```
$ gcc -Wall -ansi -pedantic p1.c -o p1
$ ./p1 3 1 2 7 4 0 2
7 4 3 2 0 2 1
Heap test success!
(program ends)
```

A few notes and restrictions regarding your implementation:

- You are NOT allowed to call the C standard library quicksort or any other sorting function from within your `heapify()` function.
- You must implement an efficient heapify algorithm. Doing a full sort on the list (via an insertion sort or a quicksort algorithm, etc) will not count for credit, since these are less efficient than simply creating a heap.
- p1.c is the starting point for your file. You should only alter the `heapify()` function and the `allocate_array()` function. Do not alter `test_heap()` or `main()`. Besides those restrictions, you are welcome to implement any additional helper functions you want (for instance, a `swap()` function may be useful when implementing `heapify()`).

2 2D Array - p2.c (10 points)

Write a function that returns a pointer to a pointer to a double where the pointer points to a valid address of m pointers-to-double each of which points to a valid address of n doubles. You should implement this function using $m + 1$ calls to `malloc()`.

```
double **dmatrix_non_contiguous(size_t m, size_t n);
```

Then, write a test main that uses your dmatrix function to allocate an array of dimension $n \times n$ (inputted via command line, `argv[1]`). Have your test main fill this array with sequential values beginning with 0. Then, print these values to `stdout` in matrix format. Specifically, a sample run should look like:

```
$ gcc -Wall -ansi -pedantic p2.c -o p2
$ ./p2 4
0.0 1.0 2.0 3.0
4.0 5.0 6.0 7.0
8.0 9.0 10.0 11.0
12.0 13.0 14.0 15.0
(program ends)
```

In p2.c, you have been given a test main and function prototype that you should fill in.

3 2D Contiguous Array - p3.c (10 points)

Redo the previous problem now enforcing that the entire allocated block of data be contiguous in memory – i.e., that each allocated block of size n doubles is adjacent to its neighbor block. You do not need to enforce that the pointer data also be in the same contiguous block as the actual floating point data. Your function should therefore use no more than two calls to `malloc()`. The prototype should be:

```
double **dmatrix_contiguous(size_t m, size_t n);
```

I.e., for an array of 4×4, we should be able to access the last element with your returned double ** pointer using 1-D or 2-D as:

```
double ** ptr = dmatrix_contiguou(4, 4);
ptr[3][3] = 15.0;
if( ptr[3][3] == 15.0 && ptr[0][ (4*4) - 1 ] == 15.0 )
        printf("Passed Contiguous check\n");
```

Above, we can see that both 2D indexing (ptr[3][3]) and 1D indexing (ptr[0][(4*4) - 1]) both work correctly, showing that our matrix is indeed contiguous.

Now, write a test main that uses your dmatrix function to allocate an array of dimension $n \times n$ (where n is inputted via command line, `argv[1]`). Have your test main fill this array with sequential values beginning with 0. Then, print these values to standard out in matrix format in using both 2D and 1D indexing.

Specifically, a sample run should look like:

```
$ gcc -Wall -ansi -pedantic p3.c -o p3
$ ./p3 4
Matrix with 1D indexing:
0.0 1.0 2.0 3.0
4.0 5.0 6.0 7.0
8.0 9.0 10.0 11.0
12.0 13.0 14.0 15.0

Matrix with 2D indexing:
0.0 1.0 2.0 3.0
```

```
4.0 5.0 6.0 7.0
8.0 9.0 10.0 11.0
12.0 13.0 14.0 15.0
(program ends)
```

In p3.c, you have been given a test main and function prototype that you should fill in.

Note that just because your code passes the above test once, does not necessarily mean that it will always pass the above test for all allocations sizes and on all systems during all system states. Be sure that your code will result in a contiguous allocation 100% of the time rather than relying on multiple subsequent calls of `malloc()` to (by chance) return contiguous allocations. That is, make sure that your function makes at most two calls to `malloc()`.

4 3D Contiguous Array - p4.c (10 points)

Redo the previous problem for a pointer to pointer to pointer to double (3D array of doubles) as:

```
double ***d3darr_contiguous(size_t l, size_t m, size_t n);
```

The allocation must be contiguous, therefore, your function should make no more than three calls to `malloc()`. I.e., for an array of $3 \times 3 \times 3$, we should be able to access the last element with your returned double *** pointer using 1-D or 3-D as:

```
double *** ptr = d3arr_contiguous(3, 3, 3);
ptr[2][2][2] = 26.0;
if( ptr[2][2][2] == 26.0 && ptr[0][0][ (3*3*3) - 1 ] == 26.0 )
        printf("Passed Contiguous check\n");
```

Now, write a test main that uses your `dmatrix_contiguous()` function to allocate an array of dimension $n \times n \times n$ (n inputted via command line, `argv[1]`). Have your test main fill this array with sequential values beginning with 0. Then, print these values to standard out in matrix format using both 3D and 1D indexing.

Specifically, a sample run should look like:

```
$ gcc -Wall -ansi -pedantic p4.c -o p4
$ ./p4 3
Matrix with 1D indexing:
0.0 1.0 2.0
3.0 4.0 5.0
6.0 7.0 8.0

9.0 10.0 11.0
12.0 13.0 14.0
15.0 16.0 17.0

18.0 19.0 20.0
21.0 22.0 23.0
24.0 25.0 26.0

Matrix with 3D indexing:
0.0 1.0 2.0
3.0 4.0 5.0
6.0 7.0 8.0

9.0 10.0 11.0
12.0 13.0 14.0
15.0 16.0 17.0
```

```
18.0 19.0 20.0
21.0 22.0 23.0
24.0 25.0 26.0
(program ends)
```

In p4.c, you have been given a test main and function prototype that you should fill in.

Note that just because your code passes the above test once, does not necessarily mean that it will always pass the above test for all allocations sizes and on all systems during all system states. Be sure that your code will result in a contiguous allocation 100% of the time rather than relying on multiple calls of malloc() to (by chance) return contiguous allocations. That is, make sure that your function makes at most three calls to `malloc()`.

5 Array Performance - p5.c (10 points)

Write a program that tests the performance difference between three different 2D array of doubles implementation:

1. Statically allocated 2D array
2. Dynamically allocated 2D array, distributed through memory (Problem 2)
3. Dynamically allocated 2D array, contiguous in memory (Problem 3)

Let the dimensions of each array be $m = n = 1000$, so that each array will have 10^6 elements. Initialize your 2D arrays to contain random numbers between 0 and 1.0. Pass the arrays to the provided kernel functions (defined in p5.c) and time each array. Do not time the initialization stages for the array, just the work stage. You may find functions in the 'time.h' library to be helpful. In p5.c, we have given you the framework of a test main function, along with the "work" kernel functions that you will use for your testing. You will need to alter the test main to add in your array allocations, initializations, timing, and results report. You are encouraged to copy your `dmatrix_non_contiguous()` and `dmatrix_contiguous()` functions that you implemented in previous problems into p5.c.

Once you've completed your implementation, test your code on your system and report your results in your `README.txt` file.

6 File Read In - p6.c (10 points)

Write a function named `read_file()` that reads a text file into an array of strings. In C, one way to implement this is as an array of array of char of fixed length. This is declared as

```
char text[MAX_LINES][MAX_CHAR_PER_LINE];
```

Where `MAX_LINES` and `MAX_CHAR_PER_LINE` are `#define` preprocessor macros.

Write a test main that tests `read_file()` and prints contents to standard out:

```
$ gcc -Wall -ansi -pedantic p6.c -o p6
$ ./p6 <input_file_name>
(input file printed here)
(program ends)
```

For the purposes of this problem, please `#define MAX_LINES` to 1000 and `MAX_CHAR_PER_LINE` to 1000.

You have been given a test main in p6.c. Your job is to implement the `read_file()` prototype. Your function should return the (integer) number of lines read from the file.

7 Alphabetizer - p7.c (10 points)

Write a function that sorts lines of text into alphabetical order. The sort must be done by swapping the pointers themselves and not moving or copying the actual text. The function prototype must be:

```
void alphabetize(char *text[], int nlines);
```

Write a test program to read lines from a file, use alphabetize, and send the alphabetized list to stdout. You are encouraged to copy your `read_file()` function in from the previous problem to re-use for the read in portion of this program. You have been given a framework for a test main and prototype in p7.c, and a test file in "students.txt".

You can assume a maximum number of 1000 lines and 1000 chars per line.

For example, if the input file "students.txt" reads:

```
Fan, Aaron
Sturt, Adam
Chakravarty, Arnav
Ali, Asad
Sandman, Benjamin
Cunningham, Bryce
Braun, Joseph
Mai, Conway
Du, Mu
Tong, Jiayang
Yang, Jinpu
O'Harrow, Joseph
Liang, Junchi
Mandava, Mayank
Zhang, Mengyu
Atanasov, Pero
Kramer, Scott
Liu, Sen
Zhu, Wanqi
Warner, Stephanie
Wu, Yichen
```

Running your code should look like:

```
$ gcc -Wall -ansi -pedantic p7.c -o p7
$ ./p7 students.txt
Ali, Asad
Atanasov, Pero
Braun, Joseph
Chakravarty, Arnav
Cunningham, Bryce
Du, Mu
Fan, Aaron
Kramer, Scott
Liang, Junchi
Liu, Sen
Mai, Conway
Mandava, Mayank
O'Harrow, Joseph
Sandman, Benjamin
Sturt, Adam
```

```
Tong, Jiayang
Warner, Stephanie
Wu, Yichen
Yang, Jinpu
Zhang, Mengyu
Zhu, Wanqi
```

8 Repeat Counter - p8.c (10 points)

Write a program that lists all the words in an ASCII text file that contain a repeated letter (two consecutive occurrences of same letter). Include the number of occurrences of each word. Note that the file can contain any words (some may be capitalized) as well as the common punctuation marks '.', '!', ',', '?', ';', ':'. The file may also contain any number of newline characters. Your analysis program should be insensitive to punctuation marks and capitalization. E.g., "Alpha" and "alpha!" should both be considered an occurrence of the same word. An example using the included "words.txt" file is given below:

```
$ gcc -Wall -ansi -pedantic p8.c -o p8
$ ./p8 words.txt
class : 1
commodo : 3
convallis : 1
dignissim : 2
efficitur : 1
fringilla : 1
habitasse : 1
massa : 3
mattis : 1
mollis : 2
nulla : 9
pellentesque : 3
porttitor : 1
sollicitudin : 2
suspendisse : 1
tellus : 5
ullamcorper : 4
viverra : 1
(program ends)
```

You may assume that the maximum number of words in a file is 2000, and each word will have at most 50 characters.

You have been provided an empty main in p8.c to build off of, as well as a test input file "words.txt". You may implement any additional functions you need, or are free to implement all functionality in `main()` itself.

9 Vector - p9.c (10 points)

In this problem, you will be implementing a simple vector data type using the framework defined in p9.c. A vector is an array that can automatically re-size itself so as to make room for appended or inserted items. The vector data type will be defined in terms of a struct and accompanying functions that will operate on it. You must implement the full API, as defined here:

```
typedef struct{
        double * data; /* Pointer to data */
        int size;      /* Size of the array from the user's perspective */
```

```
        int capacity;  /* The actual size of allocation "data" pointer points to */
} Vector;

/* Sets the element at given index of the vector to value */
void set( Vector * v, int index, double value );

/* Returns the value at given index of the vector */
double get( Vector * v, int index );

/* Doubles the capacity of the vector */
/* If capacity is 0, increases capacity to 1 */
/* Used by the "insert" and "append" functions */
void resize( Vector * v );

/* Moves all values at and after the given index to the right
 * one element, then sets the value at given index. */
void insert( Vector * v, int index, double value );

/* Appends a value to the vector, automatically
 * resizing the vector if necessary by calling the
 * resize() function */
void append( Vector * v, double value );

/* Deletes the given index, and moves all values after
 * the index to the left by one element.
 * This reduces the size of the vector by 1, but the
 * capacity should remain the same */
void delete( Vector * v, int index );

/* Initializes the vector to a specified size.
 * Capacity is set to be equal to the size of the vector.
 * All elements of the vector are initialized to 0. */
void init( Vector * v, int size );

/* Free's any memory allocated to data pointed of the vector,
 * sets size = capacity = 0, and the data pointer to NULL. */
void free_vector(Vector * v );

/* Prints the vector in a clean format. If vector is empty,
 * just print "<  >" */
void print(Vector * v );
```

You do not need to check for user errors, you should assume that the functions will be used as intended. For instance, in the `insert()` function, you can assume that a legal index will always be given, the user will not give indices less than 0 or greater than `size`.

In p9.c, the `print()` function and a test main have been defined for your use. When the test main is run, you should get the following output:

```
$ gcc -Wall -ansi -pedantic p9.c -o p9
$ ./p9
test init...
< 0.0, 0.0 >
test append...
< 0.0, 0.0, 0.0 >
< 0.0, 0.0, 0.0, 0.1 >
```

```
< 0.0, 0.0, 0.0, 0.1, 0.2 >
< 0.0, 0.0, 0.0, 0.1, 0.2, 0.3 >
< 0.0, 0.0, 0.0, 0.1, 0.2, 0.3, 0.4 >
test set...
< 0.0, 0.0, 1.0, 0.1, 0.2, 0.3, 0.4 >
< 2.0, 0.0, 1.0, 0.1, 0.2, 0.3, 0.4 >
test get...
get value = 0.10
test insert...
< 2.0, 0.0, 2.0, 1.0, 0.1, 0.2, 0.3, 0.4 >
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.2, 0.3, 0.4 >
test delete...
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.3, 0.4 >
test append...
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.3, 0.4, 0.0 >
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.3, 0.4, 0.0, 0.2 >
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.3, 0.4, 0.0, 0.2, 0.4 >
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.3, 0.4, 0.0, 0.2, 0.4, 0.6 >
< 2.0, 0.0, 2.0, 1.0, 0.1, 3.0, 0.3, 0.4, 0.0, 0.2, 0.4, 0.6, 0.8 >
test free...
<  >
(program ends)
```

10 Code Cleanliness & Coding Habits (10 points)

- Your code should be submitted with the same filenames and directory structure as the cloned github repository, all contained within a single zip file.
- We should be able to compile all problems using the included makefile with gcc.
- We are not grading to any particular coding standard and we do not have any strict requirements in terms of style. However, your code should be human readable, which means it should be very clean, well commented, and easy for the grader to understand.