Programming project#0

IT383 Spring 2023

2/2/2023

Due: Thursday, Feb. 9 at 11:55PM

Total points: 40 programming project points

Programs containing compilation errors will receive failing grades. Those containing run-time errors will incur a substantial penalty. You should make a serious effort to complete all programs on time. You will lose 10% per day (up to 3 days) if the submission is made after the deadline. Programming assignments are individual. You should complete them with your own effort. If you need help, you should come to my office hours or contact me by email. You may discuss concepts with others in the class, but not specific program code**.** The total points of this programming project are 40 programming points.

NOTE: Program 0 should be done WITHTOUT any collaboration with any other student! I am required to report any cheating in programming assignment to Dean of students.

NOTE: Each program should be placed in its own subdirectory. Also, you need to include a Makefile in the same directory as the source code to allow the instructor to compile your program automatically. You will submit a single "ZIP" file that includes all the subdirectories where your programs are located. The name of ZIP file should be *YourLastName_YourFirstName_Program0*.zip. You will need to submit your SINGLE zip file to ReggieNet.

- You might want to use the following Linux command to create a single zip file:
  % zip –r **Y*ourlastname_yourFirstName_Program0*.zip** ./1 ./2 ./3
  (e.g., Doe_John_Program0.zip)

NOTE: DO NOT upload any of your programming assignments to any public Internet repository including Github.

[1] (10 points) C program for File I/O - version 1

Create a subdirectory called "1". Change the current directory to "1".

Write a C program called "my_copy_text_count.c". This program makes a duplicate copy of a given ***text file***. Note that the source filename and destination filenames are given as command-line arguments as shown in the following example. This program also prints out the total number of characters, total number of lines, and total number of words copied from the source text file to the destination file. An example usage of the program is as follows. (Note that $ is a shell prompt, which might be different from the prompt of the shell you are using.)

$ **my_copy_text_count** /usr/share/dict/american-english ./dictionary.txt
/usr/share/dict/american-english was copied to ./dictionary.txt successfully.
Total number of characters: 971578
Total number of words: 102305

~~Total number of lines: 102305~~

Requirements:

0) Note that the source filename should be the first command-line argument.

1) **You need to use ALL of the following C APIs (all of them!)** in your program to read and write data from/into files.

fopen(), fclose(), (fgets() or fread()), and fwrite()

DO NOT use fgetc(), getline(), fputs(), read(), write(), fprintf(),or fscanf() in this program.

2) Your program **should check the error codes** returned by calls to fopen/fread/fgets()/fwrite functions.

3) **The size of the buffer (i.e., array) for file input and output operations should be limited to 1024 bytes**. **You need to use a loop in which**:

a) a block of data is read from the source file to the buffer.

b) data in the buffer is written into the destination file.

c) **the *maximum* number of bytes to be read or written specified as a parameter for fread(), fgets(), and fwrite() should be exactly 1024 bytes.**

If you use a buffer of size larger than 1024 or if you do not use a loop for reading or writing data from/to files, you will lose at least 30% of your points allocated to his program automatically.

Please note that the size of a source file can be possibly very large (as large as hundreds of Gigabytes).

4) You need to try the following sample test case to make sure that your program runs correctly. Note that "diff" is a Linux command to check if two files are exactly the same in terms of size and contents.

[a sample text file]

$ **my_copy_text_count** /usr/share/dict/american-english ./dictionary.txt
/usr/share/dict/american-english was copied to ./dictionary.txt successfully.
Total number of characters: 971578
Total number of words: 102305
~~Total number of lines: 102305~~

% diff /usr/share/dict/american-english ./dictionary.txt

⇨ Diff command should indicate that two given files are the same for each execution. Otherwise, **my_copy_text_count** program didn't work correctly!.

Hint: DO NOT ignore the return values from fopen(), fread()/fgets(), and fwrite().

Hint: Make sure that the total number of bytes that you read from the input file is same as the total number of bytes that you write to the output file. You should be careful about the invisible characters such as '\n' and '\0'.

Note: Create a subdirectory called "1". Change to the subdirectory. "% **my_copy_text_count.c**" and Makefile should be placed under the subdirectory "1".

[2] **(10 points) C program for File I/O – version 2**

Create a subdirectory called "2". Change the current directory to "2".

Write a C program called "my_copy_textbinary_count.c". This program makes a duplicate copy of a given ***text file or binary file***. Note that the source filename and destination filenames are given as command-line arguments as shown in the following example. This program also prints out the total number of bytes from the source text file to the destination file. Example usage of the program is shown in the following. Note that $ is a shell prompt, which might be different from the prompt of the shell you are using. Also, note that cmake is a "binary" file.

$ **my_copy_textbinary_count** /usr/share/dict/american-english ./dictionary.txt
/usr/share/dict/american-english was copied to ./dictionary.txt successfully.
Total number of bytes: 971578

$ **my_copy_textbinary_count** /usr/bin/cmake ./cmake.copy
/usr/bin/cmake was copied to ./cmake.copy successfully.
Total number of bytes: 5325648

Requirements:

0) Note that the source filename should be the first command-line argument.

1) You need to use **ALL of the following C APIs (all of them!)** in your program to read and write data from/into files.

open(), close(), read(), write()

DO NOT use fopen(), fgetc(), fread(), fgets(), getline(), fputs(), fscanf(), fprintf(), in this program.

2) Your program should **check the error codes** returned by calls to open/read/write functions.

3) **The size of the buffer (i.e., array) for file input and output operations should be 1024 bytes exactly. In your program, you will need to use a loop where**:

a) a block of data is read from the source file to the buffer(i.e., array).

b) the block of data in the buffer is written into the destination file.

c) **the *maximum* number of bytes to be read or written specified as a parameter for read() and write() should be 1024.**

If you use a buffer of size larger than 1024 or if you do not use a loop for reading or writing data from/to files, you will lose at least 30% of your points allocated to his program automatically.

Please note that the size of a source file can be very large (as large as hundreds Mega bytes).

4) You need to try the following sample test cases to make sure that your program runs correctly. Note that "diff" is a Linux command to check if two files are exactly the same in terms of size and contents.

[a sample run using a sample text file as the source file]

$ **my_copy_textbinary_count** /usr/bin/cmake ./cmake.copy
/usr/bin/cmake was copied to ./cmake.copy successfully.
Total number of bytes: 5325648

% diff /usr/bin/cmake ./cmake.copy

⇨ Diff command should indicate that two given files are the same for each execution. Otherwise, your program didn't work correctly! Hint: Most likely, the size of two files will be different since you write additional character(s) in the target file(s).

Hint: Make sure that the total number of bytes that you read from the input file is same as the total number of bytes that you write to the output file. You should be careful about the invisible characters such as '\n' and '\0'.

Hint: DO NOT ignore the return values from open(), read(), and write(). Check FAQ page for the sample code using open(), read(), and write() APIs on ReggieNet.

Note: Create a subdirectory called "2". Change to the subdirectory. "my_copy2.c" and Makefile should be placed under the subdirectory "2".

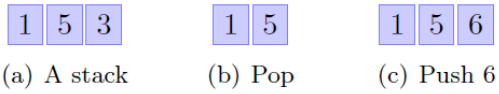
[3] (20 points) [Pointers and Arrays] C program for Stack

Create a subdirectory called "3". Change the current directory to "3".

In this problem, you will implement a basic data structures using array: stack. You can assume the elements are of type ***int*** and **the integer values that will be stored in the stack range from 0 to 10000**. A stack is usually called last in first out (LIFO) list. It supports two important operations, push and pop. Here's the list of all operations that you need to implement.

- void init_stack(stack_t *s, int capacity)
  Initializes the stack that stores integers with maximum size capacity. Y**ou should call malloc(sizeof(int)*capacity) inside stack_init())**
- int size_stack(stack_t *s)
  Returns the size of the stack, i.e., the number of elements ***currently stored/pushed*** in the array. **It should NOT return the capacity of the stack.**
- int pop_stack(stack_t *s)
  Returns the integer element on top of the stack. **If the stack is empty, the return value is undefined. (e.g., you may simply return -9999.)**
- void push_stack(stack_t *s, int e)
  If the stack is not full, push the item on top of the stack. Otherwise, do nothing.
- void deallocate_stack(stack_t *s)
  Frees this stack.
- You should NOT implement any other stack-related functions which have not been listed above.

The push and pop operations of stack only operates on one end of the array. Figure 1 shows an example.



(a) A stack (b) Pop (c) Push 6

Figure 1: Illustration of Stack Operations

Create header "dslib.h", which contains declarations of stack, and implement them in "stack.c". Test your program in "test.c", where the main function is. In test.c, you will have a main(). Inside main(), you need to test your ALL stack functions that you implemented rigorously by pushing multiple numbers into the stack and popping multiple numbers into the stack().

Also, create a "Makefile" to manage your project. For your information, you will need to use *malloc(..)* API in this program.

In dslib.h, you will need to have the following statements:

```
struct stack
{
        int count;  // the number of integer values currently stored in the stack
        int *data;  // this pointer will be initialized inside stack_init(). Also, the actual size of
the allocated memory will be determined by "capacity' value that is given as one of the
parameters to stack_init()
        int max; // the total number of integer values that can be stored in this stack
};
typedef struct stack stack_t;
```

Note: Create a subdirectory called "3". Change to the subdirectory. "dslib.h", "stack.c", "test.c" and Makefile should be saved under the subdirectory "3".

Hint: You will be able to get information on C language by reading the lecture slides on the following website: http://www.cs.cornell.edu/courses/cs2022/2011fa