Profs. Jingke Li, Classes: TR 14:00-15:50, KMC 390, Office Hours: TR 1300-1400 (in-person @FAB 120-06).

Assignment 1 Warming Up to C and Its Libraries

(Due Thursday 1/23/25)

C is a preferred language for system-level programming. This assignment aims at warming you up with C programming, with an emphasis on its library routines. You are going to implement several versions of a bubblesort algorithm. There are five separate tasks, and the assignment carries a total of 20 points.

A baseline implementation of the bubblesort is given in `bsort0.c`. Read and understand it. A test driver is provided in a separate file, `test0.c`, which accesses the sorting and printing routines through a header file `bsort0.h`.

Here is a sample of compiling and executing this program:

```
linux> gcc -o test0 bsort0.c bsort0.h test0.c
linux> ./test0
Init:   1, 3, 5, 2, 9, 6, 7, 5,
Sorted: 1, 2, 3, 5, 5, 6, 7, 9,
```

We can also create a `Makefile` with the following content:

```
CC = gcc
CFLAGS = -O -Wall
test0: bsort0.h bsort0.o test0.c
        $(CC) -o test0 bsort0.h bsort0.o test0.c
```

Then we can easily compile the program with `'make test0'`. This approach can simplify compilation when there multiple sets of programs to work with.

1. First Version (`bsort1.[ch]/test1.c`) [3 points]

The baseline implementation's test driver is rudimentary – it tests the sorting routine only with a fixed array of size 8. In this first version, you'll make two changes to the baseline implementation:

- Make a copy of the sorting program in `bsort1.c`, and add a new function, `initArray(int n)`, in it. This function creates an integer array of size `n`, and initializes it with random integer values in the range `[0,n)`. It returns the new array as its return value.
- The new driver program, `test1.c`, should prompt the user to enter an integer `n` from the command-line, which is to be used as the array size. If the entered value is not a positive integer, the program should print out a message and quit. The driver will no longer initializing the array itself; rather, it will call the function, `initArray(n)`, to do it.

You should also create a proper header file `bsort1.h` for use by the driver program.

Here is a sample execution of this new version:

```
linux> gcc -o test1 bsort1.c bsort1.h test1.c
linux> ./test1
Enter array size: 10
Init:   1, 2, 6, 3, 8, 6, 9, 8, 2, 2,
Sorted: 1, 2, 2, 2, 3, 6, 6, 8, 8, 9,
linux> ./test1
Enter array size: 0
n must be a positive integer
```

Requirements

- Other than adding a new function and necessary `#include` lines (see below), you should not change any part of the sorting program.
- You should use the C random-number generator `rand()` to produce array's initial values; use the current time (by calling `time()`) as the initial seed for the generator, so every run of the program will have a different initial array. Find out what header files are needed for these routines.
- Your program's output should match the format of the above example.

2. Char Version (`bsort2.[ch]/test2.c`) [3 points]

Copy your `bsort1.c` to `bsort2.c`. Change the array element type from `int` to `char`. This change affects all the routines in the program. The new function declarations should look like:

```
char* initArray(int n)
void swap(char* a, int i, int j)
void printArray(char* a, int n)
void bsort(char* a, int low, int high)
```

In addition, the `initArray(n)` routine should allocate an array of size `sizeof(char)*n` (instead of `sizeof(int)*n`), and initialize it with random lower-case letters (`a-z`). Here is a sample execution script of this version:

```
linux> gcc -o test2 bsort2.c bsort2.h test2.c
linux> ./test2
Enter array size: 10
Init:   k, a, s, h, s, d, t, w, s, u,
Sorted: a, d, h, k, s, s, s, t, u, w,
```

Requirements

- The only requirement is that you should not change the program structure of the routines.

3. String Version (`bsort3.[ch]/test3.c`) [5 points]

Copy your `bsort1.c` to `bsort3.c`. This third version is to sort strings of length 3. Here is a sample execution of this program:

```
linux> gcc -o test3 bsort3.c bsort3.h test3.c
linux> ./test3
Enter array size: 10
Init:   jja, igd, qwr, hos, tvi, tdz, xiu, div, ife, eoa,
Sorted: div, eoa, hos, ife, igd, jja, qwr, tdz, tvi, xiu,
```

Note that a string of size 3 can fit into the space of an integer, i.e. it is of four-byte size.

Requirements

- The program structure should stay the same.
- The `initArray(n)` routine should allocate an array of size `4*n`, and initialize it with random 3-letter strings. (It's a small challenge to generate these random strings, and to store them to array elements.)

4. File I/O Version (`bsort4.[ch]/test4.c`) [5 points]

In this version, the array is to be initialized with data from a file. Such a file contains a sequence of integer values, encoded with the standard four-byte integer encoding. For instance, a file containing the following 32 bytes:

```
a7 1b 00 00 61 0f 00 00 f1 13 00 00 59 0a 00 00
f4 02 00 00 18 03 00 00 2a 07 00 00 8e 1f 00 00
```

can be decoded as having 8 integers:

```
7079 (00001ba7) 3937 (00000f61) 5105 (000013f1) 2649 (00000a59)
 756 (000002f4)  792 (00000318) 1834 (0000072a) 8078 (00001f8e)
```

If the above file is called `in8`, then the linux command `od` can be used to show the integer values:

```
linux> od -i in8
0000000        7079        3937        5105        2649
0000020         756         792        1834        8078
```

For this version, you'll need to get familiar with C's I/O routines. We'll mention a few here.

- Reading `n` integers from a file to an array is quite easy with the `fread` function:

  ```
  fread(array, sizeof(int), n, fin);  // fin is the input file handle
  ```

- If `n` is not known ahead of time, it can be inferred from the file itself:

  ```
  fseek(fin, 0L, SEEK_END);         // go to the end of file
  int size = ftell(fin);            // this gives the file size
  n = size / sizeof(int)            // convert file size to n
  rewind(fin);                      // reset the file pointer
  ```

Your task is to write a new function `readArray(char *fname, int *np)`. It opens the input file with the given `fname`, and infers the array size `n`. It then creates an array of size `n` and reads integer values from the file to the array. Finally, it returns `n` through the pointer parameter `np`, and the array as its return value. The rest routines stay the same as in `bsort1.c`. In fact, you don't have to copy them, you can reference them with `extern` declarations, and include `bsort1.c` in the compilation process for this version.

Here is a sample script of running this version:

```
linux> gcc -o test4 bsort1.c bsort4.c bsort4.h test4.c
linux> ./test4
Enter input file name: in8
Init:   7079, 3937, 5105, 2649, 756, 792, 1834, 8078,
Sorted: 756, 792, 1834, 2649, 3937, 5105, 7079, 8078,
```

Three input files are provided to you for testing: `in8`, `in16`, and `in32`.

Requirements

- If the input file fails to open, the program should print out a proper message and terminate.

5. Combined Driver (`bsort.h/testall.c`) [4 points]

Now, we want to consolidate all tests into a single driver program. To do this, write a new `testall.c` program. It should prompt the user to select a version to run, and based on the response, it'll dispatch the control to the proper routine. The program should loop back and keep running until the user chooses to terminate.

Here is a sample script of running this driver:

```
linux> ./testall
Select version: 1(int), 2(char), 3(str), 4(file), 0(quit) 1
Enter array size: 8
Init:   1, 6, 7, 4, 2, 5, 6, 7,
Sorted: 1, 2, 4, 5, 6, 6, 7, 7,
Select version: 1(int), 2(char), 3(str), 4(file), 0(quit) 2
Enter array size: 6
Init:   d, a, m, g, u, c,
Sorted: a, c, d, g, m, u,
Select version: 1(int), 2(char), 3(str), 4(file), 0(quit) 3
Enter array size: 4
Init:   gxk, sjl, awb, yrc
Sorted: awb, gxk, sjl, yrc
Select version: 1(int), 2(char), 3(str), 4(file), 0(quit) 4
Enter the source file name: in8
Init:   7079, 3937, 5105, 2649, 756, 792, 1834, 8078,
Sorted: 756, 792, 1834, 2649, 3937, 5105, 7079, 8078,
Select version: 1(int), 2(char), 3(str), 4(file), 0(quit) 0
```

In addition to the driver program itself, you need to write a proper header file, `bsort.h`, to allow the driver to access all versions of the sorting routine. You are likely to encounter a name-conflicting issue – multiple files have routines with the same name. To fix this, you'll need to do some renaming. Think of a consistent naming convention for the different versions, and make sure they still run correctly with their individual driver.

Requirements

- Your program's interface should match the above sample.
- You need to write a `Makefile`, to help compiling this program. With the makefile, you should be able to build the combined driver with

  ```
  linux> make testall
  ```

Submission

Include your name in a comment block at the top of each program file. Zip all the source programs (`.c` and `.h` files, plus `Makefile`) into a single zip file `assign1sol.zip`, or use the linux tar utility to combine them into a single tar file, `assign1sol.tar`:

```
linux> tar cvf assign1sol.tar *.c *.h Makefile
```

(Do not include `.o` or executable files!).

Upload the file `assign1sol.[zip|tar]` on Canvas through the "File Upload" tab in the "Assignment 1" folder. (You need to press the "Start Assignment" button to see the submission options.)

Important: Keep a copy of your submission file in your folder, and do not touch it. In case there is any glitch in Canvas submission system, the time-stamp on this file will serve as a proof of your original submission time.