ECE 2220: System Programming Concepts **Fall 2016**
Machine Problem 6 **Due: 11:59 pm, Thursday, November 3**

The goal of this machine problem is to learn to use function pointers, structure pointers, passing structures and pointers to structures to functions, dynamic memory, passing binary data between programs using Shell I/O, and use of the library function **`qsort()`** with function pointers and casts. In this lab, each student is to complete two programs called **`geninput6.c`** and **`lab6.c`**. Initial versions of both programs are provided. The first program, **`geninput6.c`**, generates a collection of records in which the fields are randomized. The output of this program is used as input to **`lab6.c`**. The second program takes as input from **`stdin`** a collection on binary data using a specified structure and sorts the data on one of the fields in the structure. Finally the binary data is printed as ASCII text to **`sdtout`**.

Input

These programs are to use Shell I/O redirection techniques. The **`geninput6.c`** program creates binary output to **`stdout`** using the following format. The first 4 bytes are the number of records, and the remaining bytes the details of each record.

```
 struct Record {
    int id;
    float value;
    int a[4];
    char name[8];
};
```

where the values for each must be initialized as given below:

- **`id`** – a unique value for each record (this is provided and should not be changed)
- **`value`** – A floating point number with a random value x, such that $0.0 \leq x < 1000.0$
- **`a[4]`** - An array of four integers, each with a random value from {0, 1, 2, … , 9}.
- **`name[8]`** - A random number (2 to 7) of random characters that are upper case letters (‘A’ to ‘Z’)

The **`id`**’s have already been implemented in the provided template for **`geninput6.c`**. Your assignment is to create the randomized data for the other fields using the **`drand48()`** library function. Additional details about **`drand48()`** are given in the notes section below. The **`geninput6.c`** program takes two command line arguments: the first is the number of records to generate and the second is the seed for the random number generator. Note the most of the **`geninput6.c`** code is complete; you just fill in the details for generating the randomized data.

The **`lab6.c`** program gathers input from **`stdin`** as binary data. The data is redirected from a file or by piping directly from the **`geninput6.c`** program. You should assume that there are no errors in the input stream for simplicity. The program should gather all of the data and dynamically allocate an array of structures using the form **`struct Record`** shown above. Additional input is collected from the command line. The output from this program uses both **`stdout`** and **`stderr`**; the resulting sorted data is converted in ASCII and sent to **`stdout`** (which can be redirected to a file) while the time required to sort the data is sent to **`stderr`** (which is displayed on the terminal).

The **`lab6.c`** program accepts command line arguments specifying the field in the structure that should be used for sorting, and the type of sort.

The field in the structure which should be used for sorting is specified with one of these values:

1 Sort the structures by id.
2. Sort the structures by value.
3. Sort the structures by array.
4. Sort the structures by name.

The second argument specifies if bubble sort should be used (if the value is **2**) instead of the default sorting method of **`qsort`** (if the value is **1**). If the type of sort is **0**, then the data is not sorted but just printed.

After collecting all the input, if the program is directed to sort the data , then use the **`clock()`** command to measure the time required to sort the data, and print the result (in milliseconds) to **`stderr`**. In addition, print all of the data to **`stdout`** in text format instead of binary format. Examples of how to use **`clock()`** and print the output are provided in the **`lab6.c`** template. While the **`lab6.c`** template file has a few examples of parts of the code, you must develop most of the details.

The program must have the following supporting functions which **`qsort()`**, bubble sort, and a validation function will use. The functions specify how to compare two records.

- **`IDCompare()`** compare two structures by their id numbers. This function is provided.
- **`ValueCompare()`** compare two structures by their values.
- **`ArrayCompare()`** compare two structures by their **`a[]`** values by finding the smallest value in each array. If the smallest values are equal, compare the next smallest values. Two records are equal only if all four numbers are identical.
- **`NameCompare()`** compare two structures by their Names

Examples for comparing two records using **`ArrayCompare()`**.

#1: **`sa->a = 6, 2, 1, 6`**
`sb->a = 5, 3, 2, 9` → **`sa is smaller than sb`**

#2: **`sa->a = 4, 2, 1, 4`**
`sb->a = 9, 3, 2, 1` → **`sb is smaller than sa`**

#3: **`sa->a = 9, 2, 1, 4`**
`sb->a = 1, 4, 2, 9` → **`sa and sb are equal`**

It is **critical** that when you print the records at the end of **`lab6.c`** that you have **not** changed the values in any of the records. In particular, you are **not** permitted to re-order the numbers found in the **`a[]`** array of a record.

You code must be able to handle all combinations of redirection, such as

Generate 5 binary records and simply print them as ASCII characters. The records are not sorted:
`./geninput6 5 321 | ./lab6 1 0`
Generate 10 records and store them in a file. The file contains binary data
`./geninput6 10 123 > file.data`
Using the binary data in **`file.data`**, sort the **`value`** field with qsort. Store the records in a file.
`./lab6 2 < file.data > file.txt`
Generate 10,000 records. Use bubble sort on the array field **`a[]`** and store in a file
`./geninput6 10000 231 | ./lab6 3 2 > file.txt`
Generate 1,000,000 records. Use qsort on the string field **`name`** and discard the standard output

```
./geninput6 1000000 312 | ./lab6 4 1 > /dev/null
```

Your bubble sort implementation should be able to sort about 10,000 records in about 1 second, while **qsort** should be able to handle lists at least 100 times larger! Notice that the output for very large lists should be directed to **/dev/null**. This tells the system to delete the output data instead of wasting disk space storing the file.

For bubble sort, just like **qsort**, you must have a single function that accepts a function pointer to determine the field to use in sorting. Your prototype of bubble sort must be:

```
void bubblesort(struct Record *ptr, int records,
      int (*fcomp)(const void *, const void *));
```

where **ptr** is the pointer to the array of Record structures, **records** is the count of the number of records in the array, and **fcomp** is the function to compare records. For an additional example see the validate function.

After each call to qsort or bubblesort, you code **must** call the validate function. This function is used during grading and verifies that the list is in sorted order and contains all of the elements. The function is provided and you just need make sure it is called after the completion of any sort.

Performance Evaluation

For each of the 2 sorting algorithms and each of the 4 field values, determine the largest list size that can be sorted for each option in less than one second. In the comments at the top of your **lab6.c** file, you must have a table showing this list size for each of the 8 combinations.

Notes

1. The **drand48()** function.

Note, the random number generator `rand()` has very poor statistical properties and should never be used. Instead use `drand48()` and `srand48()`, which are included in the C standard library `stdlib.h`.

`srand48`(*x*) initializes (or *seeds*) the sequence of numbers, where *x* is a 32-bit integer. This function should be called one time at the start of the program. Setting the seed ensures that the same sequence of numbers will be produced by `drand48()` each time the program is run. To get a different sequence of numbers change the seed.

`drand48`(void) returns a double in the range [0.0, 1.0), and has the property that the number is uniformly distributed over the range. To generate a number in the range [0.0, 1000.0) use:

```
float f = 1000 * drand48();
```

to create a integer in the range {0, 1, 2, …, 9} just take advantage of type conversion:

```
int i = 10 * drand48();
```

2. The **clock()** function

To measure the performance of a sorting algorithm use the built in C function `clock` to count the number of cycles used by the program.

```
#include <time.h>
clock_t start, end;
double elapse_time;  /* time in milliseconds */
```

```
start = clock();
// call qsort or bubble sort here
end = clock();
elapse_time =  1000.0 * ((double) (end - start)) / CLOCKS_PER_SEC;
```

where `CLOCKS_PER_SEC` and `clock_t` are defined in `<time.h>`.

3. Submission requirements

The code you submit must compile using the `-Wall` flag and **no** compiler errors or warnings should be printed. To receive credit for this assignment your code must correctly sort the input using qsort for all four field values. Code that does not compile or fails to pass the minimum tests will not be accepted or graded.

4. Submit your files **`geninput6.c`** and **`lab6.c`** to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE222-1,#6. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes.

See the ECE 222 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student. See the course syllabus for additional policies.