CS-392 Systems Programming Spring 2023

Assignment 2 · Generic Insertion Sort

Lecturer: Shudong Hao *Date: See Canvas*

Preliminary

Concepts you need to understand for this assignment:

- C project file dependencies and Makefile;
- Insertion sort algorithm;
- Dynamic allocation and memory leak;
- Reading files using `FILE*`;
- Void pointers and function pointers.

Your task is to write an insertion sort that can sort any type of array.

1 File Structure

In your starter kit, you are provided with the following five files:

- `main.c`: for testing your code;
- `insertion.h`: declares data types and functions needed for insertion sort;
- `insertion.c`: implements insertion sort algorithm;
- `utils.h`: declares type-specific functions needed;
- `utils.c`: implements type-specific functions;
- `Makefile`: the makefile for compiling your code.

The red colored files are the ones you need to complete.

In the following sections, we will walk through the files thoroughly. Please read them carefully before start working on your assignment.

2 Utility Functions (`utils.h`)

It is clear that during a sorting algorithm we need to constantly compare data to decide the relative order of any two elements in the array. However, the insertion sort algorithm needs to be generic

and can deal with any type of data. Therefore, when calling insertion sort function declared in `insertion.h`, you have to create type-specific functions. In this assignment, you will need to create type-specific functions for two types: `int` and `float`:

The first function is to compare two data, and return their difference:

```
int cmpr_int(void*, void*);
int cmpr_float(void*,void*);
```

The difference should be a signed number instead of absolute value. In `cmpr_int()`, you'll compare two integers, while in `cmpr_float()`, you'll compare two single precision numbers. This function will be handy when you're calling `isort()` function.

The following print functions are to print **one** element:

```
void print_int(void*);
void print_float(void*);
```

where you print one element of an array at a time by calling `printf()` with correct format.

Lastly, we will create a function to read all the elements in an array from a file:

```
void* read_array(char* filename, char* format, size_t* len);
```

where `filename` is the name of the file where the array is stored. You can safely assume all the numbers in the file are valid, and each line stores one number. This parameter has to be passed from command-line argument `argv[1]`. If the file cannot be successfully opened, please print the following error message through `stderr` and exit with code 1:

```
File failed to open.
```

The argument `format` is passed from `argv[2]`, and can be either `"%d"` or `"%f"` for integers or floating points. The argument `len` stores the length of the array.

The function returns a pointer pointing to the array you created in this function.

In summary, the five functions you need to complete are:

- `cmpr_int()`;
- `cmpr_float()`;
- `print_int()`;
- `print_float()`;

- `read_array()`.

3 Insertion Sort (`insertion.h`)

There are two functions you need to implement for insertion sort.

3.1 Main Sorting Algorithm: `iSort()`

The function you'll need to complete is declared as follows:

```
void iSort(void* base, size_t nel, size_t width, int (*compare)(void*,void*));
```

The arguments are used this way:

(1) `void* base` : this is where you pass the base address of the array;

(2) `size_t nel` : indicates the number of elements in the array;

(3) `size_t width` : the size of each element;

(4) `int (*compare)(void*,void*)` : a function pointer pointing to a type-specific `cmpr` function in `utils.h`.

The function will perform insertion sort on the array **in place**.

3.2 Printing Array: `iPrint()`

The function `iPrint()` is to print all the elements in an array, declared as follows:

```
void iPrint(void* base, size_t nel, size_t width, void (*print)(void*));
```

where the usage of the first arguments are the same as in `iSort()`. Based on different types of the array, you should pass type-specific `print` function declared in `utils.h`.

4 Testing (`main.c`)

In `main.c` we provided a simple test case for integer case. The output should be:

```
1
3
5
10
15
20
```

You will also need to test `float` case on your own. You can modify `main.c` as you wish, but during grading we will use a separate, different `main.c` for testing. To ensure your code will work on our grading script, please follow the example provided in `main.c` closely.

You should also write a Makefile to compile your code. Make sure your `main()` function is in `main.c`.

5 Read This Before You Start (Requirements)

- `insertion.c`
  We emphasize this: because this is a generic implementation, in `insertion.c` you **must not** use `if-else` to discriminate different data types in your code. Your code must be able to deal with any data type possible. Therefore, we limit the usage of the following primitive types:
  - `int` or `size_t`: only allowed as loop iterators;
  - `char` or `u_int8_t`: can be used as you wish;
  - Types that are used in the declarations of `iSort()` and `iPrint()`: feel free to use them for their *original* purpose;
  - All other types are not allowed to appear in `insertion.c`: **no exceptions**, no matter if you actually used them, or what purpose it is. Once it appears, you will receive 30% reduction of the homework.

  You are free to create helper functions **only** in `insertion.c`; you must declare and implement them in the `.c` files instead of the header files. Do not change any of the header files.
- `utils.c`
  You are free to use any data type you like in this file. However, other than the five functions declared in `utils.h`, do not create any other functions. Do not change any of the header files.
- Other requirements:
  - Write the pledge and your name at the top of *every* file submitted;
  - No memory leak. The array you're going to sort will be freed by us in the testing script, so you don't need to worry about that. However, there must be no other memory leak, and when we free the pointer returned by your `read_array()`, there must be no errors;
  - Must be able to compile and execute without crashing (including frozen, seg fault, *etc*);
  - You do not need to include any other header files. However, if you do, make sure you can compile it successfully from your Makefile;
  - When in doubt (*e.g.*, "Can we assume...?"), please do not hesitate to ask. Pay attention to Canvas announcement closely, as we will only announce any errors/updates there without updating this document.

We hope this assignment will answer the most popular question in class: "Why do we need function pointers?!" ☺

Deliverable

Zip the following files and submit on Canvas:

(1) `insertion.c`;
(2) `utils.c`;
(3) `Makefile`.