# Problem Set 4: Sorting and Searching Structures

Please send back via NYU Brightspace

- A zip archive named as
  `PS04_Last_First.zip`
  Where `First` and `Last` are your first and last name.
  And the zip archive contains your **`sort_struct.c`** file.

Look up how to use any of the functions you might want to use at https://www.cplusplus.com/. Look at example code to determine what `#include <>'s` are needed in addition to `#include <stdio.h>`.

Download the file PS03.zip from NYU Classes.

## Problem 1

**Sort an Array of Structures**
**Total points: 50**

Points are awarded as follows:

- **10 Points** – read in struct Student data from input file
- **20 Points** – sort data using qsort() (10 points for each sort-on key)
- **10 Points** – write out struct Student data to output file
- **10 Points** - clear code, sensible formatting, good comments

You are given the files **sort_main.c and sort.h**. These files are complete and do require any additional code. It is your task to create **sort_struct.c** that contains all of the functions indicated in **sort.h**, namely:

```
read_students()
sort_students()
comp_last()
comp_id()
write_students()
```

You can use the bash script **`./build_sort.sh`** as an easy way to compile your program.

**Description of sort_main.c**
File **sort_main.c** is executred as
`./sort_struct last|id ifile.csv ofile.csv`
Where arguments are:

| | |
|---|---|
| `last` or `id` | Sort-on field name |
| `ifile` | (which must be `name_id.csv`) |
| `ofile` | (which can be any file named as `*.csv`) |

If the command line arguments are incomplete, it prints a usage statement and returns -1, otherwise it parses the command line arguments.

NOTE: Run your working program using this command line:
`./sort_struct last name_id.csv sorted_name_id.csv`
And save the output file (sorted_name_id.csv) for use in Problem 2.

It opens `ifile` for reading and `ofile` for writing. In the file `name_id.csv`, each line has three fields separated by a comma with the line terminated by '\n':

```
lastName,firstName,idNumber
```

where `firstName` and `lastName` are alpha characters and `idNumber` is numeric characters.

It reads all lines in `ifile` and use each line to fill the array `struct Student sdata[]`. It prints the student data to the terminal and prints the number of students that were read in.

It sorts the array `struct Student sdata[]` according to the sort-on key `id`.

It writes the sorted data to `ofile`.

It closes the file pointers and returns.

**Description of functions you must write**
These functions will be file **`sort_struct.c.`** The function prototypes for each function are in **`sort.h,`** which must be an `#include` at the top of your file:

```
#include "sort.h"
```

**`int read_students(FILE *ifp, struct Student *sdata)`**
This function's arguments are the input file pointer and a pointer to `sdata`. It returns the number of students read.

Remember that since `sdata` is an array of structures, in this function you can use `sdata[i].last` (or `id`) to access the elements of the structure at index `i`.

For each line in the input file, your function should:

- use **`fgets()`** to read a line from the file.
- use **`strtok()`** to parse the line into `first, last` and `id`. Since `fgets()` includes the `'\n'` in each line of the file, use `",\n"` (comma and newline) as the delimiter string in `strtok()`.
- Copy the `first` and `last` name strings from `strtok()` into the corresponding elements of the struct at index i, and write the id value into the corresponding elements of the struct at index i.

After all lines have been read

- Return the number of students read

Note that in **`struct_main.c`**, `sdata[]` is declared as

```
struct Student sdata[MAX_STUDENTS];
```

so it is an error to read more than `MAX_STUDENTS` lines from the input file and write the results into sdata[]. You could use a `for()` statement with upper limit of `MAX_STUDENTS` to enforce this limit. In general, there will be fewer than `MAX_STUDENTS` lines in `ifile`, so break out of the `for()` loop when `fgets()` returns `NULL`, i.e. when there are no more lines to read.

Look up how to use `fgets()` and `strtok()` at https://www.cplusplus.com/. Look at example code to determine what `#include <>'s` are needed in addition to `#include <stdio.h>`.

**`int sort_students(char *sort_key, int num_students, struct Student *sdata)`**

This function's arguments are a pointer to the sort_key and the number of students, in sdata[]. This function in turn calls

```
qsort(&sdata[0], num_students, sizeof(sdata[0]), comp_ftn_ptr);
```

where *`comp_ftn_ptr`* is a pointer to one of the following functions:

```
comp_last(const void * a, const void * b);
comp_id(const void * a, const void * b);
```

Remember that a pointer to a function is simply the name of the function without the parenthesis and arguments. So `comp_last` is a pointer to the function

```
comp_last (const void * a, const void * b);
```

The first lines of comp_last() must cast its arguments as pointer to struct:

```
struct student *pa = (struct student *)a;
struct student *pb = (struct student *)b;
```

You finish up comp_last () with this line:

```
return(strcmp(pa->last, pb->last));
```

Which is all you need to do, since strcmp() returns -1 if the first string is alphabetically before the second, 0 if they are the same and +1 if first string is alphabetically after the second.

Look up how to use `strcmp()` and `qsort()` at https://www.cplusplus.com/. Look at example code to determine what `#include <>'s` are needed in addition to `#include <stdio.h>`.

**`void write_students(FILE *ofp, int num_students, struct Student *sdata);`**

This function's arguments are the output file pointer, the number of students in `sdata[]` and a pointer to `sdata`. It has no return value, i.e. its return value is `void`.

For every student in `sdata[]`, use `fprintf()` to write `last, first` and `id` fields to ofile in CSV format using format string:

```
"%s,%s,%d\n"
```

Also print the line to the terminal.

## Problem 2

**Searching an array of structures**
**Total points: 50**

Points are awarded as follows:

- (10 points): Parsing command line and reading data
- (15 points): Linear Search
- (15 points): Binary Search
- (10 Points): Clear code, sensible formatting, good comments

In this problem set, you will write a program that searches a directory containing 500 names and returns the information associated with a last name entered at the terminal. Your program will implement two search methods:

- Linear Search
- Binary Search

This scenario is somewhat of a "toy" problem, in that a 500-name directory (of name and ID) is small, and the user must enter the exact last name string. However, it is sufficient to illustrate how the three searching methods work.

Your program will be structured into 3 files:

- **`search.c`** (portions provided by instructor, **portions to be completed by you**) This provides a framework that can be used for all searching methods.
- **`search.h`** (provided by instructor – no need to add anything) parameters and data structures that are used by all files in the program.

You can use bash script **build_search.sh** to compile this program.

## Parsing command line and reading data

Your program must have its command-line arguments as follows:

```
./search_struct linear name_id.csv
```

Or

```
./search_struct linear sorted_name_id.csv
```

The character indicates a choice amongst possible arguments.

**`linear`** specifies linear search
**`binary`** specifies binary search
**`name_id.csv`** is a file supplied by instructor containing an ASCII text in comma separated values format as <Last>,<First>,<ID_Number> per line in the file.
**`sorted_name_id.csv`** is the name_id.csv file that is sorted by last name, created by YOU using your sort_struct program of the previous problem.

**Parse the command line**. Write code in **`search.c`** that parses the command line.

- If your program is executed without any command-line arguments, or with an incorrect number of command-line arguments, it should print a "usage" message and then exit.
- Use variables search_method and ifile to save pointers to the command line parameters:
  search_method = argv[1];
  ifile = argv[2];
- Open the input file. Print an error with a diagnostic message if the file cannot be opened.

**Initialize data structure**. Your **main()** contains a data structure declaration:

```
struct Directory directory[DIR_LEN]
```

Initialize the data structure so that all strings are empty. For each entry in `directory[],` do the following:

```
directory[i].first[0] = 0;
directory[i].last[0] = 0;
directory[i].phone[0] = 0;
```

**Read phone book file.** Read each line from the input file and enter the data into the phone book, one line per struct array element. The phone book file is comma separated value format as <Last>,<First>,<PhoneNumber>. Note that in this struct phone number is a string. Use `fgets(), strtok() and strcpy().`

Look up these functions at https://www.cplusplus.com/. Look at example code to determine what #include <>'s are needed in addition to #include <stdio.h>.

**Print phone book** to terminal screen.

## Linear Search

Implement linear search, by filling in code for the function:

```
int linear_search(char *target, struct Directory *directory, int num_entries)
```

The first argument, `target`, is the last name to search for, provided by the user in the terminal, and the second and third arguments specifies the phone book array and its length.

When an entry is found, return the index of the entry. In addition, be sure to treat the case in which the entry is not found, in which case the function should return a value of -1.

Note that you can use `strcmp()` to compare two strings (target and last). Remember that its return value `status = strcmp(target, last)` indicates:

| | |
|---|---|
| -1 | target is alphabetically before last |
| 0 | strings are the same |
| 1 | target is alphabetically after last |

## Binary Search

Implement binary search, by filling in code for the function:

```
int binary_search(char *target, struct Directory *directory, int num_entries)
```

This function uses the same arguments as for linear search, but requires that the phone book is ordered by the last name. The sorting is already implemented in the provided **`main.c`** by calling **`qsort()`** and using **`strncmp()`** in the sort comparison function.

When an entry is found in **`binary_search(),`** return the index of the entry. In addition, be sure to treat the case in which the entry is not found, in which case the function should return a value of -1.

See lecture slides for a description of the binary search algorithm.