The Objectives of this lab are the following:

- C's array/pointer duality
- Understand the use of make and makefiles
- Casting pointers in C (specifically to and from void*)
- Defining, passing, and calling functions by pointer

As was the case for labs 1 through 4, ask the TAs lots of questions. They are there to help. And, **you CAN work with peers in completing the labs. This is a bit tricky when everybody is remote; you may use slack, email, discord, etc. Up to you.** But, in no circumstances should you SHARE code, which involves emailing code and then submitting it as your own.

Overview

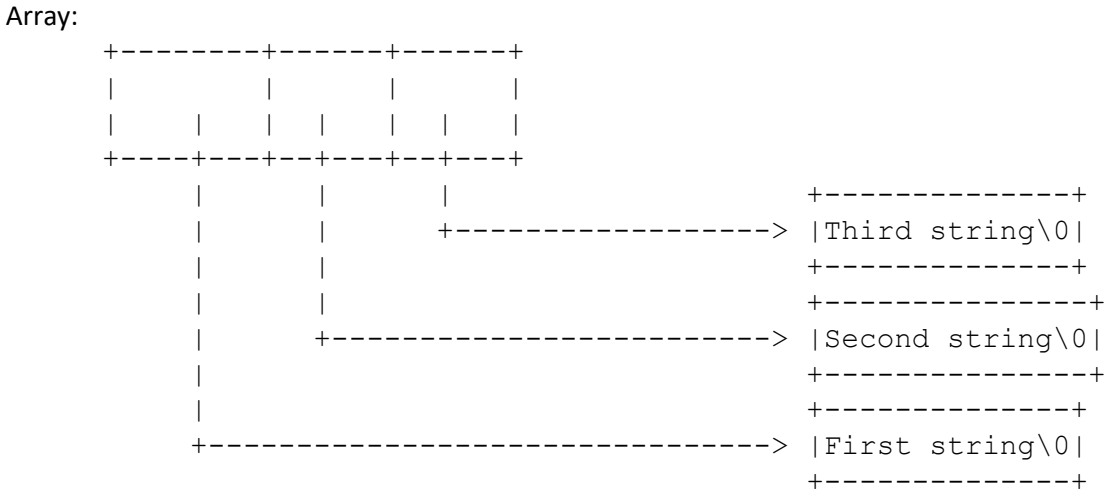
This lab asks you to complete a generic sorting algorithm by adding a third, function parameter to an existing function, and update the two test programs that you are being provided to use your new parameter. The below instructions walk you through the first two parts of this lab: completing the `sort_array` function and updating `sort_strings`.

Specifics

The goal is to develop a general purpose sorting routine, that is capable to sort using **any** data type. The generalized function is defined in *sort.h* and implemented in *sort.c*. Get these functions from

```
/home/jagodzf/public_html/teaching/csci247/labs/lab4
```

The array to be sorted contains `void*`s. A `void*` is simply a pointer to an unknown type. Let's say that I have an array of strings, for example. The actual array is an array of `char*`s and would look like:



The array has size 3, but each string has a different length. For the most part, we don't care what the pointers point at.

Here is the problem: we care what the pointers point at regarding ordering. The only way I can compare `void*`s is by where they point in memory. This is not helpful. Pointers into the .text section would sort before .data or the stack. I need a generic way to compare two unknown data types.

Consider the example above: an array of strings. I could add a compare-function for strings in *sort.c*. This function would look something like this:

```
static
int ordered_strings (void* left, void* right)
{
   char* left_string = (char*) left;
   char* right_string= (char*) right;
   return strcmp (left_string, right_string) < 0;
}
```

All this function does is cast the `void*`s to `char*`s. We happen to know that this is correct in this example. The actual comparison is performed by the C-library function `strcmp`.

The test in *sort_array* needs updating. Currently, the test looks like this:

```
if ( Array[i] < Array[i+1] ) {
  swap(&Array[i], &Array[i+1]);
  have_swapped = 1;
}
```

See how it compares the addresses? For strings, the test would look like this:

```
if ( ordered_strings (Array[i], Array[i+1]) ) {
  swap(&Array[i], &Array[i+1]);
  have_swapped = 1;
}
```

This will work for strings, but defeats the goal; this is not a generic sorting function.

Instead of adding every possible ordering of every data type to *sort_array*, we can pass a comparison function as an argument. Technically we cannot pass a function as an argument, but we can pass a pointer to a function. This syntax, however, is strange.

Right now, *sort_array* looks like this:

```
void sort_array(void* Array[], unsigned size);
```

We need to add a third parameter, let's call it `ordered`. In C, we usually declare the type followed by the name (e.g., void* Array[], unsigned size, etc.). The name of the function pointer is in the middle and has a weird set of parentheses. The new declaration of `sort_array` looks like this:

```
void sort_array(void* Array[], unsigned size, int (*ordered)(void*,void*));
```

See what I mean about the strange syntax! The parentheses around the symbol bind the asterisk to the name rather than to the return type. Otherwise, it would be a function that returns an `int*`.

Using the function is straight-forward. The test now looks like this:

```
if ( ordered(Array[i], Array[i+1]) ) {
  swap(&Array[i], &Array[i+1]);
  have_swapped = 1;
}
```

We can use the function parameter like any other function, but only within *sort_array*.

Now the caller must provide an array, its size, and a function that determines if the two -- whatever-they-are – are in the correct order. Let's fix `sort_strings`. Right now, it only passes the first two arguments to `sort_array`. We need to create the third argument, a function that compares two strings, but we have already done this. However, we put it in the wrong file. Move `compare_strings` from *sort.c* to *sort_strings.c*. Now we can correct the call to `sort_array` in main:

```
sort_array((void**)data, data_size, &ordered_strings);
```

Makefiles and `make`

Provided in the directory named above is the file Makefile. The utility `make` is the command-line program which issues compilation instructions that are specified in the file Makefile.

The use of a Makefile streamlines the compilation process. Since we have only fixed sort_strings, asking make to build everything would create a bunch of errors. There are many details to using make, which we don't cover in detail here. For the time being, just know that a Makefile includes variables names, and individual instructions for different pieces of code that you want compiled. In the Readme file that you are given, there are multiple compilation instructions, including `all`, `sort.o`, `check`, etc, and including `clean`. Each of these is accompanied by instructions of what the compiler should do IF the user invokes that compilation option. We can build just *sort_strings* with the following `make` command:

```
$make sort_strings
```

What does `clean` do? And `all`? Ask your TA! Hint : issue `make clean`, look at the directory structure, and then issue `make sort_strings`, and look at the directory. What has changed?

And we can use *sort_strings* to test:

```
$./sort_strings
"rodeo" and "reading" are out of order
"reading" and "polish" are out of order
"polish" and "lima" are out of order
"lima" and "job" are out of order
```

Hmm, something went wrong. I think we sorted in the wrong direction. One of your tasks is to fix this.

The second set of tests is `sort_struct`. This one is slightly more complicated. Rather than strings, it is an array of pointers to structures. Create an ordering function such that `sort_array` orders the cars in ascending order or years.

What you must do

- Finish implementing the generic `sort_arrays` function
- Update `sort_strings` to sort in alphabetic order
- Update `sort_structs` to sort in increase model year order.

Important: Do not modify the for-loop in the main functions. It is there to ensure that the arrays are sorted in the correct order.

Rubric and submission

Please upload the following files to canvas

- *sort.c*
- *sort.h*
- *sort_strings.c*
- *sort_structs.c*

Submit the asked-for files as the Lab 4 Submission item on Canvas. You must submit your file by the due date/time specified on Canvas.

Sorting strings correctly	5 points
Sorting cars correctly	7points
Coding style	3 points
	15 points