CMSC 313 — Spring 2017
Project 4 — Sorting C Structs

Assigned	Thursday, March 30th
Program Due	Wednesday, April 5th by 11:59pm
Updates:	None yet.

Objectives

The objectives of the programming assignment are 1) to review your basic C/C++ programming skills 2) to practice using C structs 3) to practice basic C pointer operations 4) to explore how C structs are actually implemented, via assembly language

Background

Structs and Field Offsets

Your program for this project will work with an array of structs that are defined in a C program. Each member of the array is a struct that contains data about an e-book. The data type for that struct is declared with:

```
#define AUTHOR_LEN      20
#define TITLE_LEN       32
#define SUBJECT_LEN     10

struct book {
   char author[AUTHOR_LEN + 1];    // first author
   char title[TITLE_LEN + 1];
   char subject[SUBJECT_LEN + 1];  // Nonfiction, Fantasy, Mystery, ...
   unsigned int year;              // year of e-book release
};
```

In memory, each `struct book` occupies a contiguous block of memory. However, it is not necessarily the case that each field of a `struct` is placed right after the previous one. The reason is that the C compiler sometimes adds padding between fields of a struct to align each field to its natural boundary (as discussed in class). This can be observed through the `gdb` debugger. For example, let us define a simple 2-field `struct rec`, and then declare an instance of the struct, as follows:

```
struct rec {
   char field1[5];
   int field2;
};
```

```
struct rec myRec = {"Hi", 47};       // A global instance of a struct rec
```

If we examined this structure in gdb:

```
(gdb) print myRec
$1 = {field1 = "Hi\000\000\000",
  field2 = 47}

(gdb) print &myRec
$2 = (struct rec *) 0x804a3a0

(gdb) print &myRec.field1
$3 = (char (*)[5]) 0x804a3a0

(gdb) print &myRec.field2
$4 = (int) 0x804a3a8
```

Notice that field2 starts 8 bytes after the start of field1, not just 5 as you might expect.
Using this kind of exploration with `gdb`, we can discover the actual offsets of all the fields of any `struct`, and use that information to then define constants for our assembly code, thus:

```
; Offsets for fields in struct rec.
;
%define FIELD1_OFFSET 0
%define FIELD2_OFFSET 8
```

For this project, you will add the following to your assembly code, replacing the "???" with actual numbers you got from exploring the `struct book` struct much as we did with the `struct rec` in the example above.

```
; Offsets for fields in struct book.
;
%define AUTHOR_OFFSET ???
%define TITLE_OFFSET ???
%define SUBJECT_OFFSET ???
%define YEAR_OFFSET ???
```

Once we have discovered and defined the offset values, we can use them in our assembly code to access the individual fields in any struct of that type. For example, if the ESI register holds the address of an instance of `struct book`, then `[ESI+YEAR_OFFSET]` can be used to reference the `year` field of the `struct`. Similarly, `[ESI+ECX+SUBJECT_OFFSET]` can be used to access the *i*-th character of the subject string where the value of *i* is stored in ECX.

Selection Sort

In this project, you will also be implementing a simple sorting algorithm: Selection Sort, to sort an array of `struct book`s. The algorithm is described in the following pseudocode:

```
selection_sort(VALUE vals[], int count) {
```

```
        for (i = 0; i < count - 1; i++) {
            min = i;
            for (j = i + 1; j < count; j++) {
                if (vals[j] LESS THAN vals[min]) {
                    min = j;
                }
            }
            if (min != i) {
                SWAP vals[i], vals[min];
            }
        }
    }
```

In plain English: the algorithm iterates over each position in the list, comparing all elements at that position or after, looking for the lowest value among the remaining elements. It then swaps the lowest found into the current position if it's not already there. It is one of the simplest algorithms, and also requires less swapping, which fits our needs.

Assignment

You will be writing a mixed-language program, mostly implemented in the C language, that sorts an array of `struct book`s. It will implement the majority of the Selection Sort algorithm, but will call out to a small subroutine to compare each pair of books, which you will write in assembly language.

Your program will be executed with an input file which will be read in from standard input (we will use the shell's I/O redirection to to this, so you will not have to explicitly open any files in your code). A large sample data file is available at:

```
/afs/umbc.edu/users/p/a/park/pub/cmsc313/spring17/proj4/books.dat
```

You need to copy this file into your own directory.

Your C program's first task will be to process the input, filling in an array of `struct book` structs in memory. The array will be dimensioned to hold a maximum of 100 books. Your program will read in a line at a time, each line comprising a record of a single book. It will continue reading records until either (a) it hits the end-of-file, or (b) it fills the entire array, whichever comes first. Here are a few sample records from the file:

```
Breaking Point, Pamela Clare, Romance, 2011
Vow, Kim Carpenter, Nonfiction, 2012
1491, Charles C. Mann, Nonfiction, 2006
Three Weeks with My Brother, Nicholas Sparks, Nonfiction, 2004
```

(Note that the fields in the records are not in the same order as the fields in your struct--that should not matter.)
Each of the string fields (title, author, subject) in your struct must be null-terminated--that is the reason each is dimensioned as the length limit + 1. So, AUTHOR_LEN, for example, is the actual maximum number of real (i.e., non-null) characters that can be in the author's name, including spaces.

You can use `scanf()` to read in the fields. Every record is "clean", meaning all fields are present and of the right format. The only place where a comma (',') appears is as a field separator (i.e., it does not appear embedded in a title, author name, etc.). For most of the fields, you can, and should, read directly into the actual field in the struct book. However,

there is one complication: The title in the file record might be too large to fit in the space allotted in the corresponding field in the struct. (All other fields are guaranteed to fit.) Here is an example of a too-long title:

```
Harry Potter and the Chamber of Secrets, J.K. Rowling, Fantasy, 2012
```

This title is 39+1 characters long including the null. We've only allocated 32+1 characters for The title in our struct. If you try to read this title directly into the struct's field, one of two things will happen:

- fscanf() you will read in more characters than will fit in the field, overrunning the field, with bad consequences; OR:
- you use the field width specification in fscanf() to stop after the a limited number of characters, in which case the rest of the title will stay in the input buffer, waiting to be (erroneously) read as part of the next field

Neither of those two options are satisfactory. So, you will have to first temporarily read the title from the file into a buffer large enough to hold all of it (the provided code implies that the title will never be more than 80 characters long, and you can depend on that), and then only copy a limited number of chars into the actual struct field (make sure you set aside room for the null!).

Once you've read in all of the book records, you will sort the books, based primarily on year of publication, oldest year first. For all books published in the same year, you will subsort by title, in alphabetic order (don't be scared--see the Hints section). See the partial sample output below.

You will sort the entries in the array by implementing Selection Sort, as described in the Background section above. Since it is a simple array, iterating over it should be easy. To swap two members, you can take advantage of the fact that struct-to-struct assignment copies the entire contents of the struct. Slightly inefficient, compared to using pointers, but that's why we chose Selection Sort, to reduce the number of swaps.

Within the inner loop of your Selection Sort implementation, you need to compare two records. To do this, your sort implementation ***must*** call the assembly routine `bookcmp`, which you will implement in bootcmp.asm. Since you have not yet learned how to pass parameters in using C's call-by-value mechanism, you will store ***pointers*** to the two `struct book` records you want to compare in global pointers called `book1` and `book2`.

You will implement `bookcmp` in assembly code, in the file `bookcmp.asm`. It will look in the global variables `book1` and `book2` for pointers to the two records to be compared. It will return one of the integer values -1, 0, or +1 in the register EAX, depending on whether book1 is strictly less than, equal to, or greater than book2, respectively.

You will obviously have to implement your C code and assembly code in separate files. You should call the C file `sort_books.c`, and your assembly file `bookcmp.asm`. The entry point of your assembly subroutine must be called `bookcmp`. Your sorting function in sort_books.c must call this `bookcmp` subroutine. We have not yet learned how C passes its parameters, so for now, your C code should put pointers to the two `struct book`s to be compared in the global pointers "book1" and "book2", so that they can be accessed easily from the assembly code. Thus, one of your first instructions in `bookcmp` should be (after any required saving of registers with PUSH):

```
mov     ebx, [book1]
mov     ecx, [book2]
```

Now, EBX and ECX will contain pointers to the two `struct book`s to be compared, and you can use indexed addressing modes on these.

At the end of your program, you will print out all of the books in sorted order, in the exact same format as the original

input.

When you build your program, you must compile or assemble each source file separately:

```
linux2% gcc -m32 -ansi -Wall -g -c sort_books.c
linux2% nasm -f elf -g bookcmp.asm
linux2% gcc -m32 sort_books.o bookcmp.o -o sort_books
```

(We will provide a Makefile template appropriate for this project in the same directory as sort_books.c.)

Then, you can run the executable `sort_books.out` file produced. A sample run should look like:

```
linux2% ./sort_books.out < books.dat
Dead until Dark, Charlaine Harris, Fantasy, 2001
Three Weeks with My Brother, Nicholas Sparks, Nonfiction, 2004
1491, Charles C. Mann, Nonfiction, 2006
Eclipse, Stephenie Meyer, Romance, 2007
New Moon, Stephenie Meyer, Romance, 2007
<... more output here>
```

Note: The graders will use a different data file to test your program from what is provided with this project spec. You are encouraged to modify the data file yourself to check additional cases.

Implementation Notes/Hints

- An important part of this project is using an appropriate indexed addressing mode in your assembly code to access the fields in the struct. Think this through carefully. A clean and logical approach to this problem will yield clean and logical code that is easier to construct and, more importantly, easier to debug. You will be graded on how well you use indexed addressing modes.
- Your program should be reasonably robust and report errors encountered (e.g., empty array) rather than crashing.
- When subsorting by title, the strings should be compared using dictionary ordering. For example, any string starting with the letter 'a' comes before any string that starts with 'b' (regardless of length). Capital letters should come before any lowercase letters (this is the way the ASCII code works). In the case that one string is a prefix of another, the shorter string come first. E.g., "egg" comes before "eggs".

  Note that the strings in `struct book` are all C-style NULL-terminated strings. You will have to watch for this as you loop to compare strings for subsorting by title. One nice feature is if one title ends before the other, it's null will always be less than any real character in the other title (think about it), so the comparison will work without any special treatment.
- You may find the LEA instruction useful. (LEA = Load Effective Address.) The LEA instruction stores what would have been the address in the source operand into the destination operand. For example,

  ```
  lea     edi, [ESI+SUBJECT_OFFSET]
  ```

  will move ESI + SUBJECT_OFFSET and store it in EDI.

- The skeletal `sort_books.c` program this project is available for copying directly on the GL file system at:
  `/afs/umbc.edu/users/p/a/park/pub/cmsc313/spring17/proj4/sort_books.c`

- In order for code in other files (like sort_books.c) to call your assembly subroutine, you must declare the `bookcmp` label to be `global`. (If the label is not global, then only code from the same file can use that label.)
  Thus, you must include the following declaration in your `bookcmp.asm` file:

  ```
  global bookcmp
  ```

  The `global` declarations are typically made just after the `SECTION .text` declaration.

  Also, to tell nasm that book1 and book2 are defined elsewhere, you must have the following declaration in your assembly language program:

  ```
  extern book1, book2
  ```

  This will allow you to use `book1` and `book2` as labels for the memory locations that hold pointers to the two books to be compared.

- Conversely, in order to be able to call the subroutine `bookcmp` from your C code, you must insert the following declaration in your C code to tell the gcc compiler that `bookcmp` is defined in a different file. Otherwise, it will only look for `bookcmp` in the current file (and not find it). You must also specify details of how it expects arguments, and what it returns. To tell gcc this, include the declaration:
  `extern int bookcmp(void);`

  Also, for bootcmp.asm to be able to see and access book1 and book2 by name, you must make them global in your C code; i.e., define them in sort_books.c ***outside of any function.***

What to Submit

Before you submit your program, record a sample run of your program using the UNIX script command. You should show your program sorting the data file that we provided.

Use the UNIX `submit` command on the GL system to turn in your project. You should submit *three* files: (1) the C source code that implements the sorting, in a file named `sort_books.c`, (2) your assembly language code impementing the bookcmp function, named as `bookcmp.asm`, and (3) the typescript file of your sample run. The UNIX command to do this should look something like:

```
submit cs313_park proj4 sort_books.c bookcmp.asm typescript
```