ECS 30 **Homework #8 (50 points)** **Winter 2013**

Due: Friday, March 8th. Written: 4pm in 2131 Kemper. Programs: 11:55pm using handin to cs30, p8 directory. Filenames: main.c, main.h, file.c, file.h, search.c, search.h, vector.c, vector.h, Makefile, and readSalaries.c.

Written (7 points)

Written: (7 points) pp. 614-615: 1, 3; pp. 654-655: 4, 6; p. 695: 5, 6, 11. Zyante: 6.15, 7.1 – 7.5, 9.7

p. 614 #1 Define a structure type called `subscriber_t` that contains the components `name, street_address, and monthly_bill` (i.e., how much the subscriber owes).

p. 615 #3 From p. 613 #2, there was a typedef struct `olympic_t`, and a declaration `olympic_t competiton`. How would you call a function `scan_olympic()` passing `competition` as an output argument?

p. 654 #4 What are the characteristics of a binary file?

p. 655 #6 What is a file pointer?

p. 695 #5 Compare the execution of the macro call: `MAC(a,b);` to the execution of an analogous function call: `mac(a,b);` Which of the following two calls is sure to be valid and why? `mac(++a, b);` or `MAC(++a, b);`

p. 695 #6 When you write the body of a macro definition, where should you use parentheses?

p. 695 #11 Describe the purpose of the "**`defined`**" operator.

Programming (43 points, 45 minutes)

All programs should be able to compile with no warnings when compiled with the –Wall option. Your Makefile must use the –Wall option on all gcc lines. We will be taking points off for warnings. You should put your name(s) in a comment on the first line of each file. You will find my executables as well testing files in ~ssdavis/30/p8. User inputs are in **bold**. The prompts and output format of each program must match the examples exactly.

#1 (5 points, 5 minutes) Filename: readSalaries.c

Write a program that prints out the information of specific records stored in the binary file salaries.dat by using fseek() and fread(). Your program may not use an array of SalaryInfo. salaries.dat is created by the executable createSalaries.out that is compiled from createSalaries.c that stores the information using the SalaryInfo struct listed below. The information is read from salaries.csv. All four of these files are available in ~ssdavis/30/p8. You may assume that there will be no errors in the use of your program so there is no need for error checking of any kind.

```
typedef struct
{
  char occupation_title[31];
  int employment;
  double percent_of_total_employment;
  double median_hourly_wage;
  double mean_hourly_wage;
  double annual_mean_wage;
} SalaryInfo;
```

```
[ssdavis@lect1 p8]$ head -4 salaries.csv
Occupation title,Employment,Percent of total employment,Median hourly wage,Mean hourly
wage,Annual mean wage
All Occupations,7546400,1,27.87,34.43,71610
Office and Administrative Supp,1671920,0.2216,16.47,17.89,37210
Computer and Mathematical Occu,1174180,0.1556,37.86,39.93,83050
[ssdavis@lect1 p8]$ readSalaries.out
Please enter an index (-1 = done): 0
All Occupations: Employment: 7546400, Percent of employment 100.00%
```

```
Median hourly wage: $34.43, Mean hourly wage: $27.87
Mean annual wage: $71610

Please enter an index (-1 = done): 1
Office and Administrative Supp: Employment: 1671920, Percent of employment 22.16%
Median hourly wage: $17.89, Mean hourly wage: $16.47
Mean annual wage: $37210

Please enter an index (-1 = done): 550
Pest Control Workers: Employment: 0, Percent of employment 0.01%
Median hourly wage: $20.67, Mean hourly wage: $20.67
Mean annual wage: $43000

Please enter an index (-1 = done): 2
Computer and Mathematical Occu: Employment: 1174180, Percent of employment 15.56%
Median hourly wage: $39.93, Mean hourly wage: $37.86
Mean annual wage: $83050

Please enter an index (-1 = done): -1
[ssdavis@lect1 p8]$
```

#2 (38 points, 45 minutes) Filenames: main.c, main.h, file.c, file.h, search.c, search.h, vector.c, vector.h, Makefile

This is the second in the series of assignments that interact with genealogy files. You are to write a program that reads a GEDCOM file into two sorted arrays of structs, and then permits searching for the children of a named person as in p7. This time there will be one type of struct named Individual, that contains information about an individual, and another type of struct named Family, that contains information about a family. Besides all of the information from the two arrays of p7, the two structs contain additional information that facilitates searching using binary search. Despite these changes, the output will be identical to that of p7.

Though the two arrays of structs are a different way of storing the information of p7's four arrays, the tasks of parsing the file will not be dramatically different. However, because the struct arrays will be sorted, the accessing routines will be different. On the bright side, you will not have to write sorting and binary searching routines. Instead, you will be calling the qsort() and bsearch() functions of stdlib.h to do these chores.

Though it will involve many changes, I suggest you start with either your or my p7 code, and then modify it. You are welcome to use my p7 code without fear of accusations of plagiarism. You will note that there are fewer specifications for this assignment than in the past. It is time for you to develop a program on your own. However, I will give suggestions and guidance for dealing with the structs and stdlib functions.

Specifications:

1. All information unique to an individual must be stored in a struct of type Individual that is typedeffed in main.h.
   1.1. Individual information must include INDI (the individual's ID), name, FAMC (ID of family for which the individual is a child), and FAMS (ID of family for which the individual is a spouse).
   1.2. There will be a dynamically allocated array of Individual structs that will sorted by IDs.
2. All information about a family must be stored in a struct of type Family that is typedeffed in main.h.
   2.1. Family information must include FAM (the family's ID), HUSB, WIFE, chil_count (the number of children, an int), and CHILs as a dynamically allocated array of IDs based on the number of children. Note that the CHILs array differs from p7 in that it is a char** (a dynamic array of dynamically allocated char arrays) instead of a dynamically allocated array of chars.
   2.2. read_family() must call a new function named add_child() that will take a char* line, and a Family* that is a pointer to a single Family (not an array of Family). add_child() will create a new char** array one larger than the current CHILs to handle the additional child. There must not be a memory leak in add_child().
   2.3. There will be a dynamically allocated array of Family structs that will be sorted by IDs.
3. All remaining dynamic memory must be freed in deallocate().
4. Since all the variables in the structs, except CHILs and chil_count, have a one-to-one relationship with the tags in the files, you may use the uppercase tags for the variable names, e.g. FAMS and FAMC. All of the variables of the structs will be dynamically allocated char*, except CHILs and chil_count.
5. find_children() must call a new function named print_children() that prints the names of all the children in a specific family. print_children() must have three parameters: a Family* that is a pointer to a single Family (not an array of Family), the individuals array, and individual_count.

6. To use both qsort() and bsearch(), you will need to write two compare functions that can compare pointers of Individuals or Family structs, and return an int based on comparing their respective IDs, similar to those returned by strcmp(). (Hint: not only "similar", but actually identical.)

Suggested Order of Development

Unhappily, the conversion to the two struct arrays from the four char* arrays must be done in large chunks before you can recompile. To facilitate intermediate compiling, you will have both sets of variables passed into functions until you can eliminate the old char arrays. Nonetheless, you should be able to compile without warnings after each of the following steps. The following instructions are guidelines. I am intentionally not providing detailed instructions unless there is a new concept involved. The overall approach will be: 1) convert p7 to using the two struct arrays while also storing new information in the two struct arrays, 2) use the new information to improve the searches.

1. Write the typedeffed structs in main.h. Name them Individual, and Family. Remember to terminate the typedef with a semi-colon. Compile and run. You will be #including main.h in all files because Family and Individual will be mentioned as parameters in every file.
2. Add pointers to Individual and Family to main(), named individuals and families. Add the addresses of the pointers to the parameters of initialize(), and dynamically allocate the arrays in initialize(). To avoid later seg faults, in seta all pointers within the structs to NULL, and the child_count of each Family to zero. Compile, and run.
3. Add individuals and families to the parameters of read_file(). Pass individuals to read_indi(), and families to read_family(). Compile and run.
4. Eliminate the two old arrays from read_indi(), including its parameters, re-write read_indi() using the individuals array, and improve get_ID().
   4.1. Now that everything in the structs is dynamically allocated, and you have many variables that are based on IDs, it will be worthwhile improving the old get_ID() from p7. Instead of having get_id() just parse the ID, we can have it dynamically allocate room for the ID, copy the ID to the new char array, and then return the address of the new char array.
   4.2. Instead of indiIDs[*individual_count], you will now use individuals[*individual_count].INDI, and just assign it the address returned by the new version of get_id(), i.e., individuals[*individual_count].INDI = get_ID(line);.
   4.3. Use strstr() to find the lines with FAMS, and FAMC, then assign their respective variables with address returned from get_id();
   4.4. Instead of names[*individual_count], you will use individuals[*individual_count].NAME.
   4.5. In main(), after the call to read_file(), add a for-loop that has a printf() that will print INDI, NAME, FAMS, and FAMC of each of the elements of individuals on one line.
   4.6. Don't worry about the other parts of Individual for now. Compile and run, but don't try searching because it will no longer work. You should see the list of IDs, names, FAMS, and FAMC.
5. Though the individual IDs in smith.ged are sorted, there is no guarantee that they are. In order to use qsort(), and bsearch() search with individuals, we will need a comparison function that acts like strcmp(), but takes two const void*'s as parameters.
   5.1. In search.c, write individual_cmp() that returns the value returned by strcmp() when it compares the INDI fields of the void* passed to individual_cmp(). The only trick here is casting each void* to an Individual*, and then using "->" operator to access their INDI fields. Since casting has lower precedence than "->", you will have to use an extra set of parentheses.
   5.2. Just above your for-loop in main(), call qsort() to sort the individuals array. If you need help with the parameters of qsort(), just type "man qsort" on the command line.
   5.3. Compile and run, but don't try searching. You should see the INDI still sorted.
6. Eliminate the old p7 arrays from read_family() and read_file(), including their parameters, and re-write read_family() using the families array.
   6.1. Dealing with FAM, HUSB, and WIFEis simple now if you use get_ID().
   6.2. Because each CHIL encountered will involve increasing the size of the CHILs array by one (think back to resize() of count.c) , you must write the new function named add_child(), that takes care of the whole process of resizing, and appending the new child's ID. get_ID() again makes this task easier. You must pass add_child the address of the single current Family struct you are filling, i.e. add_child(&families[*family_count], line). This will give you a little experience working with struct pointers, and the "->" operator. Make sure you don't have a memory leak!

6.3. Don't bother with the other parts of Family for now. When you are done, you will find that read_family() looks pretty nice.
6.4. In main(), change the for-loop so that it prints the information from the families array instead of the individuals array. You should now have a printf() of FAM, HUSB, WIFE, and the chil_count of each of the elements of individuals on one line. Use an inner for-loop that prints out the CHILs IDs on the next line.
6.5. Compile and run, but don't try searching because it still will not work. You should see the list of family, husband, wives, and children's IDs. Make sure the number of children IDs matches chil_count, and those in the file.

7. As in part 5, write a family_cmp() function in search.c, and place another call qsort() in main() that sorts the families array. This time you will cast the void* to Family*.
   7.1. Compile and run, will still no searching. The list of families should continue to be sorted. When you are satisfied that all is stored and sorted properly, remove the for-loop code from main().
8. Append the two struct arrays to the parameters of find_children(). Append individuals to the parameters of find_name(). Adjust the calls accordingly. Compile and run, but don't search yet.
9. Eliminate names from the parameter list of find_name(), and rewrite find_name() to utilize the NAME of the elements of individuals. In find_children(), add a printf() after the call to find_name() that prints the index returned by it. Change the if statement in find_children() to "if(name_index >= 100)" that will ensure that it is never true.
   9.1. Compile and run. Try searching for some names. The indices printed on the screen should make sense to you. When you are satisfied, remove the temporary printf, and change the if statement back to "if(name_index >= 0)".
10. In p7, we could use the same function, findID(), to search for an ID in either spousesIDs, or childIDs. We cannot do that in p8 because those char* arrays are subsumed within the two struct arrays. We will use bsearch() to find the Family that has FAM that matches the FAMS of the named individual.
    10.1. Comment out both findID() function calls. We will not use them, but it is nice to remember where they were called.
    10.2. Declare a Family, named family, and a Family* named family_ptr in find_children(). Assign family.FAM the value of individuals[name_index].FAMS. Among the parameters for bsearch() will be the address of key_family as the key, families as the array, and family_cmp as the comparison function. Store the result of the bsearch() in family_ptr. If you need help with the parameters of bsearch(), just type "man bsearch" on the command line. What should happen if family_ptr ends up NULL?
    10.3. Compile until there are no warnings, but do not run it.
11. Time to write a brand new function called print_children(). This function will take the address of a specific Family, the individuals array, and individual_count as its three parameters, and prints the names of the children listed in the Family.
    11.1. The basic structure is a for-loop of the Family's children that contains a call bsearch() that finds the child's ID in individuals and then prints its name.
        11.1.1. For these calls to bsearch(), you will declare an Individual named individual and an Individual* named individual_ptr in print_children().
        11.1.2. As your loops works through the CHILs array of the Family*, assign the CHILs[i] value to individual_key.INDI.
        11.1.3. Among the parameters to bsearch() will be address of the individual_key, individuals, and individual_cmp. Store the value returned by bsearch() in individual_ptr.
        11.1.4. Print the NAME of individual_ptr.
    11.2. Replace the commented out call to find_ID() in find_children() with a call to print_children(). You will also be deleting a lot of extra lines in find_children() as well as many variable declarations because they are no longer needed.
    11.3. Delete both commented out find_ID() lines.
    11.4. Alter find_children() to do its test for no children so that it makes use the family's chil_count.
    11.5. Compile, and run. The search should perform as in p7.
12. Eliminate all remaining mentions of the old p7 arrays, pass the two struct arrays to deallocate(), cleanup initialize(), and rewrite deallocate() to completely free all dynamically allocated memory in the two arrays. Since you initialized all pointer variables to NULL, deallocate() should check for that before trying to free a variable.
13. Compile and run. Everything should run as in p7. You are done!