:qECS 30 **Homework #7 (50 points)** **Winter 2013**

Due: Wednesday, February 27th. Written: 4pm in 2131 Kemper. Programs: 11:55pm using handin to cs30, p7 directory. Filenames: main.c, file.c, file.h, search.c, search.h, vector.c, vector.h, Makefile

Written (8 points)

p. 695: 4, 7, 8, 9, 10, 12.

#4 How does the C compiler know whether to look for an included file in the system directory or in the program's directory?

7 What are C's five storage classes? What are the default storage classes for variables declared in each of the following environments?
declared at the top level
declared as function parameters
declared as local variables of a function

#8 What is the purpose of storage class register?

#9 Discuss this statement: If a program has five functions that manipulate an array of data values, it makes more sense to declare this array at the program's top level so that each function does not need to have an array parameter.

#10 Why is the argument value 1 used much more often than the argument value 0 in calls to the exit function?

#12 When function main() of a C program has a non-void parameter list, why is the value of its first parameter never less than 1?

Zyante: 5.9, 6.14, 6.16 – 6.18, 9.6

Programming (42 points, 65 minutes)

All programs should be able to compile with no warnings when compiled with the –Wall option, e.g. gcc –Wall taxes.c. Beginning this week we will be taking points off for warnings. You should put your name(s) in a comment on the first line of each file. When you are to write your own functions, since main() should be the first function in each file, you will need to provide a prototype above main() for each function you write. You will find my executables as well testing files in ~ssdavis/30/p7. The prompts and output format of each program must match the examples exactly. User inputs are in **bold**.

This is the first of a series of assignments that interact with genealogy files. You are to write a program that reads a **GEDCOM** (an acronym standing for ***GE****nealogical* ***D****ata* ***COM****munication*) file, and then provides the names of the children of a person named by the user. "A GEDCOM file consists of a header section, records, and a trailer section. Within these sections, records represent people (INDI record), families (FAM records), sources of information (SOUR records), and other miscellaneous records, including notes. Every line of a GEDCOM file begins with a level number where all top-level records (HEAD, TRLR, SUBN, and each INDI, FAM, OBJE, NOTE, REPO, SOUR, and SUBM) begin with a line with level 0, while other level numbers are positive integers." From http://en.wikipedia.org/wiki/GEDCOM.

For this assignment you will only be dealing with the INDI and FAM top-level records. The information for a given record ends when a tag line of another top-level record occurs in the file. Thus, everything between lines that begin with zeroes deal with a single record. An INDI record provides information about individual, including their names (NAME tag). The FAM records provide information about a family, including its spouses (HUSB and WIFE tags) and children (CHIL tags). Each individual and family has a unique ID that have a '@' at each end. These IDs are established in each INDI and FAM tag line, and used in the HUSB, WIFE, and CHIL tags lines. Note that of all the tag lines with which your program interacts, only the NAME tag line does have an ID as its data. Other than INDI and FAM tag lines, all of the tag lines your program will interact with will begin with a '1', because they need no further elaboration, unlike a BIRTH that would have a date and a place.

For this first genealogy assignment, we will be streamlining the information we store. This will make our searching inefficient, but will minimize the number of data arrays you will need to maintain. We will be using "parallel" arrays for this assignment. All of the elements at a given index of parallel arrays contain information about the same object. In this case, the indiIDs and names arrays will be parallel arrays that deal with individuals, and the spousesIDs and childIDs arrays will be parallel arrays that deal with families. For example, both indiIDs[2] and names[2] will both refer to the same individual, and both spousesIDs[5] and childIDs[5] will both refer to the same family. You will be storing multiple IDs separated by spaces in each element of the spousesIDs and childIDs.

Since GEDCOM files have different lengths, your arrays will have to be dynamically allocated. Since we are using parallel arrays, the number of elements in indiIDs and names will be the same, and the number of elements in the spousesIDs and childIDs will be the same. A simple way to determine the size needed is to quickly read the file, and only count how many INDI tags and FAM tags there are, and then allocate accordingly. After reading the file, you will rewind() it so you can read through it again and this time process its data into the arrays.

The search process relies on linear search and the parallel arrays. When given a name by the user, search the names array for that name. If it is found, the individuals ID can be found in the indiIDs array at the same index as the name. Next search the spousesIDs array for that ID. Once you find the family that has the individual as a spouse, then loop through that family's corresponding childIDs element and look for the corresponding ID in the indiIDs array. Once you find a child's ID in the indiIDs, then print their name from the names array.

Specifications

1. main.c will only contain main()
   - 1.1. main() should act as the manager of a program. It declares the variables that are shared among the top level functions, ensures that the program command is proper, and calls user functions.
     - 1.1.1. The variables will be four char** for your arrays, two ints to hold the sizes of the two sets of parallel arrays, and a FILE*.
     - 1.1.2. The name of the GEDCOM file will be passed as the command line parameter to main().
2. vector.c
   - 2.1. initialize() will allocate and initialize the values of the elements of the four arrays.
   - 2.2. deallocate() will free all dynamically allocated memory.
3. vector.h will contain the prototypes of vector.c as well as macros for constants used in vector.c.
4. file.c
   - 4.1. count_records() will try to open the file, count the families and individuals in the file, and rewind the file. If the file is unreadable, then count_records() should notify the user of the problem, and call exit().
   - 4.2. read_file() will loop through the file calling read_indi(), read_family(), or read_other() based on the record tag, until the end of the file is reached.
   - 4.3. read_indi() will process an INDI record to copy an individual's ID into indiIDs, dynamically allocate enough room for their name in names, copy their name into names, and increment indi_count.
   - 4.4. read_family() will process a FAM record to append the ID of each spouse to the family's spousesIDs, and append the ID of each child to the family's childIDs. There should be a space character between each ID.
   - 4.5. read_other() will simply loop through any record that is not a INDI or FAM record until each comes upon the another top-level record.
   - 4.6. get_ID() parses an ID out of a tagged line, and returns a pointer to it.
5. file.h will contain the prototypes of file.c.
6. search.c
   - 6.1. find_children() conducts the searching process that is explained above.
   - 6.2. find_ID() finds an ID within an array of char*s, and returns its index.
   - 6.3. find_name() will parse a name entered by the user, and then search the names array for it, and returns its index.
7. search.h will contain the prototype for find_children().
8. Makefile
   - 8.1. It will create an executable named family.out.
   - 8.2. It must create an object file for each source code file.
   - 8.3. You must use the –Wall and –g options on all lines invoking gcc.
   - 8.4. It must have a clean: option that uses rm –f to explicitly remove the files created by the Makefile, i.e., family.out, family.o, file.o, vector.o, and search.o.

Suggested Steps in Development

Obviously, this is the largest project you have had for the quarter. It is easy to feel overwhelmed, but if you use top down design techniques, I think you will find it manageable. The following steps will walk you through an iterative process where you write a little code, compile until error free, test the program using temporary printfs, and then move on to another small part of the code. By working with small parts at a time you will be able to concentrate on the task at hand without being confused by other issues, and you will know the source of errors must be coming from your current task. At the end of each step your code should compile without warnings, and run properly up to that point. At the beginning the instructions are quite detailed and thorough. As the steps continue, they become more sketchy so that you can practice what you have learned. Since the instructions are quite long, I suggest that you place check marks next to sections as you complete them.

As you go you will create "stubs" of functions. Stubs are otherwise empty functions, that may have "return 0;" statements if they are non-void.

We will be using smith.ged to test your program, and assume that you have smith.ged in the same directory as your source code.

1. In main.c, write main() with just its command line parameters, a printf() that prints argc and argv[1], and good old return 0. This should compile with no warnings. Test by providing a command line parameter, e.g. "a.out smith.ged"
2. Write a Makefile that has three pairs of lines. Remember to use –Wall and –g with gcc.
   2.1. The first pair creates family.out from main.o. At this point family.out is only dependent on main.o. You use the "-o" option of gcc to specify an executable name.
   2.2. The second pair creates main.o from main.c. You use the "-c" option of gcc to create an object file from a source code file. There is no need to use the –o here—gcc already knows the name of the object file based on the source code file name.
   2.3. The third pair removes the files created by the Makefile. The first line of the pair is simply "clean: ". The second line uses "rm –f" to remove family.out and main.o.
   2.4. Once you have written your Makefile save it. Type "make". You should see gcc create main.o, and then create family.out with no errors or warnings. Try running family.out by typing "family.out smith.ged". You should see "2 smith.ged" printed on the screen.
3. In main(), before your printf(), add a test to ensure the number of arguments is two. If it is not then report the usage of the program, and return 1.
   3.1. After writing the code, make your program. It should compile without warnings.
   3.2. Test your program by typing "family.out". The usage statement should be printed, and the program should terminate without printing anything else. Test it again with "family.out file1 file2". Again the usage statement should print, and the program should not get to your original printf statement.
   3.3. Try running family.out by typing "family.out smith.ged". You should see "2 smith.ged" printed on the screen.
4. Writing your count_records() stub.
   4.1. In main(), add declarations for the two count variables and the FILE*, write the call count_records() that passes argv[1] as well as the addresses of the count variables. Have count_records() return the FILE* it fopens.
   4.2. Create file.c with a stub for count_records(). Rather than mentioning argv in the parameter list, make that parameter "const char* filename" because that is more descriptive. Since it is non-void function, it must have a return statement. Have it "return 0;" for now.
   4.3. Create file.h with "#ifndef FILE_H", "#define FILE_H", and #endif on the three lines after the author's line. Copy and paste the heading of count_records() from file.c between "#define FILE_H" and #endif. Don't forget to add a semi-colon after the heading to complete the prototype.
   4.4. In main.c, add a #include for file.h below the #includes for the system header files.
   4.5. In the Makefile, add a pair of lines to create file.o, add file.o to the dependencies of family.out and to its gcc line, and add file.o to the list of files to be removed by clean:. Remember that file.o is dependent on both file.c and file.h.
   4.6. Type "make", and you should see file.o, main.o, and family.out all created with no errors. You will have a warning that fp in main() is unused.
   4.7. Try running your program with "family.out smith.ged". You should see "2 smith.ged" printed on the screen.
5. Writing the part of count_records() that ensures that the file is readable.
   5.1. Declare a FILE* in count_records().
   5.2. In count_records(), add a temporary printf at the top of the function that prints the passed filename from argv[1].
   5.3. In main(), remove the temporary printf() that displayed the command line parameter information.

5.4. Write the code to try to open the file for reading, and testing whether it is successful. If it is not successful, then notify the user, and exit(1). Have the return statement now return your FILE*.
5.5. Your code should now make with no warnings. Run your program, by typing "!f" on the command line. This will execute the last command you typed on the command line that began with an "f". Your program should print out "2 smith.ged," and exit normally.

6. Writing the part of count_records() that counts the records in the file.
   6.1. Write the code to count the number of times INDI and FAM are top level tags in the file. Use fgets(), and strstr() with a local char array line[80] to find files that start with a '0', and contain one of the two strings.
   6.2. Add a printf() after your fgets loop that prints out the values of individual_count, and family_count.
   6.3. Add a printf() in your main() just above the return statement that also prints out the values of individual_count, and family_count.
   6.4. Remove the temporary filename printf() statement from earlier in the function since you are sure that aspect of your code is working properly.
   6.5. Have the function rewind() the file pointer just before returning.
   6.6. Make without warnings. Don't run your program yet. You want to make an unbiased hand count first.
   6.7. Open smith.ged in vi, and determine how many times INDI and FAM occur in the file. (Remember "/" for searching in vi)
   6.8. Now run your program with smith.ged, and see if both printed statements match the counts you counted.
7. Writing initialize() stub
   7.1. Initialize() will dynamically allocate the four arrays.
   7.2. In main(), declare the four char** pointers, and write the call to initialize(). You will need to pass the two count values, as well as the addresses of all four char** pointers. This will be the only function to which you will pass the address of the char pointers.
      7.2.1. To avoid confusion during this tutorial, please name your arrays indiIDs, names, spousesIDs, and childIDs.
   7.3. Create vector.c, and write a stub for initialize(). Include vector.h below the authors' names.
   7.4. Create vector.h. Look back at the steps for creating file.h, and duplicate its format using VECTOR_H instead of FILE_H. Copy the heading of initialize() from vector.c, and paste it for use as a prototype.
   7.5. In main.c, include vector.h.
   7.6. In the Makefile, add a pair of lines to compile vector.o, add vector.o to the dependencies of family.out, add vector.o to the gcc line of family.out, and add vector.o to list of files to be removed by clean:.
   7.7. Type "make", and you should see vector.o, main.o, and family.out recreated with no errors or warnings. Note that file.o would not need to be recompiled because you have made no changes to either of its files.
8. Writing the code for initialize().
   8.1. After you have declared your local variable(s), add a temporary printf() that prints out value of the two count variables. Remove the temporary printf() in count_records() that printed out the count variables, but leave the one in main().
   8.2. Write the code to dynamically allocate the four char* arrays. indiIDs and names have the same size, and spousesIDs and childIDs have the same size. Since these are addresses of variables declared in main(), you will ALWAYS use the dereference (*) operator with them in THIS (and only this) function.
   8.3. Write the loop to initialize each of the elements of indiIDs, and names.
      8.3.1. Since ID's are never longer than 20 chars, the indiIDs array elements may be dynamically allocated 20 chars. Store the constant as macro in vector.h.
      8.3.2. Since the length of names varies, you will have to dynamically allocate the char arrays of names when you read the file the second time. You should assign NULL to each element of names as a sentinel value.
   8.4. Write the loop to initialize each of the elements of spousesIDs and childIDs.
      8.4.1. Since there are never more than two spouses, the spousesIDs array elements may be dynamically allocated 40 chars. Store the constant as macro in vector.h. You should set the first character of each element of spousesIDs to a '\0' to make its string zero length.
      8.4.2. Since the number of children are unknown, you will have to dynamically allocate the char arrays of childIDs when you read the file the second time. You should assign NULL to each element of childIDs as a sentinel value.
   8.5. On the command line type !m to recall the make command. You should have no warnings.
   8.6. Run your program, and make sure the two temporary count printf() statements agree.
9. Writing the deallocate() stub.
   9.1. Though deallocate() will be the last function called by main(), it is useful to test whether allocate() worked properly, so you will write it now while initialize() is fresh in your memory.

9.2. Write the call to deallocate in main() after the call to initialize(). deallocate() will need all of the variables of main() except the FILE*. Since deallocate() will be the last function called in main(), just pass in the values all the variables—not their addresses.
9.3. Add a deallocate() stub to vector.c, and copy its heading to vector.h.
9.4. After a make with no warnings, your program should now run, and just print the two count lines.

10. Writing the deallocate() function.
10.1. In your two for-loops that free() the char arrays, you need to test if an element of names or childIDs has been allocated by seeing if its value is NULL or not. This why you set them to NULL in initialize().
10.2. After the for loops, free() the four char** arrays themselves.
10.3. After a make with no warnings, you program should now run, and just print the two count lines.

11. Writing the read_file() and find_children() stubs.
11.1. If I have a good sense of how all of the data will flow to the functions called by main(), I normally write main() all at once, and then add the needed stubs to make it compile without warnings. So far for this tutorial, you have been writing main() piecemeal. In this section, you will complete main() by using the techniques you have already learned. Feel free to refer back to early sections for guidance.
11.2. Write the read_file() call in main(), write its stub in file.c, and add its prototype in file.h. read_file will need the four arrays, and the FILE*.
11.3. Write the find_children() call in main(), create search.c, write the stub for find_children() in search.c, create search.h including the search() prototype, update the Makefile to now compile, link, and rm search.o. find_children() will need the four arrays, and the two count variables. Since none of the variables should be changed in find_children(), they should be passed by value, not address.
11.4. After a make with no warnings, your program should run with no problem, and just print the two count lines.

12. Writing the read_file() function
12.1. The tricky part of gathering information from the file is that you don't know you've reached the end of a record until you've read the first line of the next top-level record, or reached the end of file. A simple way to handle this is to have a char array named "line" that always holds the contents of the current line in the file. read_indi(), read_family(), read_other() will have "line" passed them, and will refill it before returning. All three functions will return the returned value of fgets() to help determine when the end of file has been reached.
12.2. Write the read_file() function with its calls to the three functions determined by the tag in line.
12.2.1. You will need a couple local variables to keep track of how many individuals and families you have processed.
12.2.2. The basic structure is a loop that contains an if-else statement. You will need to "prime the pump" with a single call to fgets() before the loop.
12.2.3. Remove the temporary printfs() from main() and initialize(). Add a printf() that prints the current line as the first line within the loop.
12.2.4. Besides "line" and the FILE*, the three functions take different parameters.
12.3. Write the stubs for the three functions. Add a printf() to each stub that prints out the name of the function. Have all three functions return NULL.
12.4. After a make without warnings (that only compiles family.o and links family.out), run the program. You should see "0 HEAD", and "read_other" on the screen before the program terminates normally.

13. Writing the read_other() function
13.1. This function should simply read through the file, line by line, until it reaches a line that begins with a zero, or fgets returns NULL.
13.2. After a make without warnings, run the program. You see the "0 HEAD", "read_other", "0 @I01@ INDI", and "read_indi" before the program terminates properly.

14. Writing the read_indi() function
14.1. Copy the code from read_other() as a starting point.
14.2. Before the loop, copy the individual's ID into indiIDs using individual_count as an index. I used strrchr(), then strchr(), and finally strcpy() to make it so I only stored the ID without the @'s.
14.3. Inside the loop, use strstr() to detect the line that contains NAME. Because we will later print the name, you will need to parse the line so that name looks more presentable. Construct the name in a local char array using a loop with strtok(), and sprintf(). You can make life easier if you use multiple characters in the delimiter string of strtok(). After creating the name in your local char array, allocate an array based on the length of the name, store the address of the array into an element of names, then strcpy() the name into the element, and then increment individual_count.

14.4. At the end of read_file() , after the loop, add a for-loop that prints out all of the contents of names, and indiIDs on one line for each index value.

14.5. After a make without warnings, run the program. You should see "0 HEAD", "read_other", "0 @I01@ INDI", "read_indi", "0 @I02@ INDI", …, "0 @I43@ INDI", "read_indi", "0 @F01@ FAM", "read_family", "I01 Edwin Michael Smith" followed by the rest of the contents of the two arrays, before the program terminates properly. Compare the latter output with that in the smith.ged file. If the print out is wrong, go back into read_indi() and fix the problem(s) before continuing.

15. Writing the read_family() function

15.1. Copy the code from read_other() as a starting point.

15.2. Inside the loop use strstr to select lines with HUSB, WIFE, and CHIL.

15.2.1. If HUSB, or WIFE, then separate out the individual ID, without their @'s. Wait a minute, you just did this in read_indi(). You could copy the code from read_indi(), but since you expect to need to do this with the CHIL tags too, why not write a little helper function? Quickly create get_ID() that takes line as its parameter, that replaces the second '@' with a '\0' and returns a pointer to the beginning of the ID. Update file.h, and replace the appropriate code in read_indi() with a call to get_ID().

15.2.1.1. Use sprintf() to append the returned pointer from get_ID() (with a trailing space) to the spousesIDs element based on the family_count as an index. Note that this works well the first time because you set the first character to '\0' in initialize().

15.2.2. If CHIL, then you will need to expand the array in the childIDs element based on the family_count as an index.

15.2.2.1. First, call get_ID() and store its returned value in a local char* variable, named ptr .

15.2.2.2. If the childIDs element is NULL, then use malloc() with strlen() to set the childIDs element to the address of an allocated string based on the new child's ID. Use strcpy() to copy the new child's ID. Beware! If you try to free() a NULL pointer you will get a seg fault!

15.2.2.3. If the childIDs element is not NULL, you will need to use another local char* variable named temp to store the address returned from a call to malloc() that allocates based on the length of the old childIDs element and the length of the new child's ID. Use one sprintf() to copy the old childIDs element followed by the new child's ID into temp, free() the old childIDs element, and set its value to that in temp.

15.3. Before the return statement, increment family_count.

15.4. Remove the printfs() from read_indi().

15.5. Change the for-loop at the end of read_file() to print out the contents of childIDs and spousesIDs.

15.6. After a make without warnings, run the program. You should see "0 HEAD", "read_other", "0 @I01@ INDI", …"0 @I43@ INDI",…"0 @F01@ FAM", "read_family",…, "0 @F15@ FAM", "read_family", "0 @S1600@ SOUR", read_other, …, "0 TRLR", "read_other", "Sp: I01 I40 Ch: I42 I41", followed by the rest of the list of spouses and children before the program terminates properly. Compare the output with that of the real smith.ged file. If the print out is wrong, then go back into read_family() and fix the problems before continuing.

16. Double checking read_file()

16.1. Before starting on the search routine, you need to make sure that main() is getting its four arrays correctly filled.

16.2. Cut the for-loop from the end of read_file() that printed the spouses and children, and paste it in main() after the call to read_file().

16.3. Type "make clean", and after it has removed all of the object files and the executable, type "make" to ensure that all is well with the whole project.

16.4. Run the program. Your output should be identical to what you saw at the end of step #15. If not, then determine which variable as fault, and fix it in the read_file() function.

16.5. Alter the main() for-loop to print out the indiIDs and names arrays, make, run, compare with the real file, and repair as needed.

16.6. At this point, you should be confident that all four arrays are properly filled. To avoid confusion, you should now remove all the temporary printfs in file.c as well as the for-loop in main().

17. Writing find_children()

17.1. The searching process is described just before the specifications. Though I will describe the writing of find_children() as a unit. You would be wise to add temporary printfs() after writing each loop, making, and running the program to check the values produced.

17.2. The search function should continue until the user enters "Done" as the name to be sought.

17.3. Though it is tempting to read the name using scanf(), since the number of names entered varies, you will have to use fgets() to read a whole line of text, and then strtok() to separate the parts into separate names. To use fgets() with the keyboard, you use predefined FILE * "stdin", e.g. "fgets(line, 80, stdin);"

17.4. Surprisingly, the hardest part is parsing the name provided by the user, and finding it in the nameIDs array. We can postpone dealing with this by creating function stub for find_name() that takes a name, the names array, and individual_count as parameters, and returns the index of the name in the names array. It will return -1 if it cannot find the name, and should notify the user. Call find_name() from find_children(), create it's stub, and update search.h.

17.5. Since names and indiIDs are parallel arrays, if the name was found at index i in names, then that person's ID is in indiIDs[i]. Call find_ID() with spouses_IDs, family_count, and the ID from indiIDs at the found index. Write a stub for find_ID(), and update search.h. If find_ID() returns -1, then report that the individual never married. Have your find_ID() stub return -1. This would be a good time to make, and test your program.

17.6. Since spousesIDs and childIDs are parallel arrays, we can use the index returned by our call to find_ID() to access the IDs of the children of the family for which the individual was a spouse. Copy the element of the childrenIDs to a temporary char array so can use strtok() with the temporary array. Write a loop that will go through the temporary array, and parses out each child's ID, then calls find_ID() with that ID, and then uses the index returned by find_ID() as an index in names to printf the child's name.

17.7. After a make without warnings, run the program and enter some names. Since the find_name() stub always returns -1, your program should always report that the name was not found. You will note that the name has a '\n' at its end which causes an unexpected line change on the screen. We will take care of that in find_name().

18. Writing find_name()

18.1. Before writing the search code, eliminate the '\n' at the end of the name, by replacing the '\n' with a '\0' using either strlen(), strchr(), or strtok().

18.2. find_name() will look for an entry that contains all names entered by the user as individual strings using strstr(). This will allow for erratic spacing, missing middle names, and names out of order. For example, "Smith, Keith Lloyd" will be parsed into "Smith", "Keith", "Lloyd", and your search routine will look for a names entry that contains all three strings in any order, e.g. "Keith Lloyd Smith".

18.3. You will use a space character, a comma in your strtok() delimiter string.

18.3.1. If you have a local array of five char*'s named name_ptrs, then you can simply assign the addresses returned from strtok() into name_ptrs using a incrementing index variable until strtok returns NULL.

18.4. Write a loop that goes through names array looking for the contents of name_ptrs using strstr(), and returns the current index when it finds a match for all of the names stored in name_ptrs in one names element. If it complets the loop without a match, then tell the user, and return -1.

18.5. After a make without warnings, run the program and enter some names. Your program should now report if the name is not found, else that the name was not married because the find_ID() stub returns -1.

19. Writing find_ID()

19.1. At this point you should be able to write this linear search using strstr() without help.

19.2. After a make without warnings, run the program and enter some names. If you have no errors, you should be done!

In case you are interested, here are the meanings of the tags in the GEDCOM files we will be using:

Tag	Meaning
ADDR	Address
AFN	Ancestral File Number
AUTH	Author
BAPM	Baptism
BIRT	Birth
BURI	Burial
CALN	Call number within repository.
CHAN	Change
CHAR	Character set or encoding
CHIL	Child
CHR	Christening
CONC	Concatenate lines
CONT	Continue on next line.
CORP	Corporation
DATA	Data
DATE	Date
DEAT	Death
DEST	Destination system

DIV	Divorce
EDUC	Education
ENGA	Engaged
FAM	Family
FAMC	Child within "family"
FAMS	Spouse within "family"
FILE	Filename
FORM	Format
GEDC	GEDCOM details
HEAD	Header
HUSB	Husband
IMMI	Immigration
INDI	Individual record
LABL	Label
MARR	Marriage
MEDI	Media
NAME	Name
NOTE	Note
NSFX	Name suffix

OBJE	Object
OCCU	Occupation
PAGE	Page number
PHON	Phone number
PLAC	Place
PUBL	Publication
REFN	Reference number
REPO	Repository
RESI	Residence
SEX	Gender
SOUR	Source
SUBM	Submission
TEXT	Text
TIME	Time
TITL	Title
TRLR	Trailer
VERS	Version
WIFE	Wife

```
[ssdavis@lect1 p7]$ cat smith.ged
0 HEAD
1 SOUR FTW
2 VERS 8.0
2 NAME Family Tree Maker for Windows
```
Edited by Sean
```
4 LABL Marriage Ending Status
0 @I01@ INDI
1 NAME Edwin Michael /Smith/
2 SOUR @S1600@
1 SEX M
1 BIRT
2 DATE 24 MAY 1961
2 PLAC San Jose, Santa Clara Co., CA
1 OCCU
2 PLAC Software Engineer
1 EDUC
2 DATE BET. 1979 - 1984
2 PLAC UC Berkeley
1 _DEG
2 DATE 1984
2 PLAC B.S.E.E.
1 FAMS @F01@
1 FAMC @F02@
0 @I02@ INDI
1 NAME Alice Paula /Perkins/
```
Edited by Sean
```
0 @I43@ INDI
1 NAME Charles /Neilsen/
1 SEX M
1 BIRT
2 DATE 26 JUN 1918
1 DEAT
2 DATE 17 JUL 1971
2 PLAC Hayward, Alameda Co., CA
1 CHR
2 DATE 10 JUL 1996
2 PLAC Community Presbyterian Church,
Danville, CA
1 FAMC @F08@
0 @F01@ FAM
1 HUSB @I01@
1 WIFE @I40@
1 CHIL @I42@
2 _FREL Adopted
2 _MREL Adopted
1 CHIL @I41@
2 _FREL Natural
2 _MREL Natural
1 MARR
2 DATE 27 MAY 1995
2 PLAC San Ramon, Conta Costa Co., CA
1 ENGA
2 DATE 5 OCT 1994
2 PLAC San Francisco, CA
0 @F02@ FAM
1 HUSB @I18@
1 WIFE @I02@
1 CHIL @I29@
2 _FREL Adopted
2 _MREL Adopted
1 CHIL @I01@
2 _FREL Adopted
2 _MREL Adopted
1 MARR
2 DATE 4 JUN 1954
2 PLAC Sparks, Washoe Co., NV
2 SOUR Hannah was the widow of James
Matthewson.
0 @F03@ FAM
```

```
[ssdavis@lect1 p7]$ family.out
Usage: family.out <ged_filname>
[ssdavis@lect1 p7]$ family.out smith.ged smith.ged
Usage: family.out <ged_filname>
[ssdavis@lect1 p7]$ family.out nofile.ged
Unable to read nofile.ged.
[ssdavis@lect1 p7]$ family.out smith.ged
Please enter a name (Done = exit): Edwin Michael Smith
Mason Michael Smith
Amber Marie Smith
Please enter a name (Done = exit): Jefferson, Elna
Hanna Smith
Ingar Smith
Ingeman Smith
Martin Smith
Please enter a name (Done = exit): Hanna Smith
Hanna Smith never married.
Please enter a name (Done = exit): Astrid Smith
Astrid Smith had no children.
Please enter a name (Done = exit): Betsy Jones
Betsy Jones not found.
Please enter a name (Done = exit): Done
[ssdavis@lect1 p7]$
```