

CS 2213, Spring 2009  
Assignment 5: Phone Book 1  
Due: 5pm, Monday, March 23, 2009

*Goal:* The purpose of this assignment is to learn to use dynamically allocated memory and structures.

*Objectives:* be able to write code that reads data from files (`fopen()`, `fclose()`, `fgets()`); be able to write code that uses `malloc()` and `realloc()` to dynamically allocate memory; be able to write code to implement a dynamic array list; be able to write code to use/manipulate an array of structures; be able to write code to identify whether two strings match.

### Phone Book Application

Write a program `phonebook1` that will read in a file containing pairs of lines names and phone numbers, place those into a dynamic array list, and then read in a list of names (one per line) from the standard input. For each name read from the standard input, the program should determine if there is an exact match in the dynamic array list. If there is a match it should print out the name, followed by the phone number. If there is not a match, it should print out the name followed by the string "No match found."

### Input/Output Specification

The program should work with input and produce output exactly as described here. (Automated tools that do a character by character comparison may be used for grading.) For inputs that do not comply with this specification, your program may print an error message and exit; there should not be any inputs which would cause your program to have undefined behavior.

*phonebook1 command line.* The program executable should expect as a command-line argument the name of a file containing a list of phone numbers. Example:

```
./phonebook1 phonenumber.txt
```

*data file.* The file specified on the command-line (e.g., `phonenumber.txt` will contain lines of text of up to 80 non-null characters. (If there are more than 80 characters—not counting '\n'—on a line, the additional characters should be silently discarded, producing the same result as if all of the lines were truncated to 80 characters.)

Each pair of lines will represent one 'phonebook entry'. The text of odd lines will be considered to be a person's name. The text of even lines will be considered to be the phone number of the person named on the previous line. The program should treat these items as uninterpreted strings of up to 80 non-null characters. (In particular the program should **not** try to parse the

name to determine the last name or parse the the phone number to encode the individual digits.) The entries will not necessarily be sorted in any way.

An example data file is:

```
John Smith
+49 1234 5-67890
Jeff
458-5667
Abe Lincoln
1-888-WIT-HOUS
```

*Standard Input.* The program should interpret each line on the standard input as a name that it should attempt to find in the list. (If there are more than 80-characters + '\n' on a line, the additional characters should be discarded.) The program should stop when it sees a blank input line or EOF.

An example of some input to the program would be:

```
Jeff
Dr. von Ronne
Supercalifragilisticexpialidocious
John Smith
Abe
Jeffery

Bubba
```

For testing, you may wish to redirect the input from files containing lists of names such as that found in this example.

*Standard Output.* For each of the lines on the standard input, the program should attempt to find a phonebook entry who's name is an exact match of the name in the standard input line. If there is a match, the program should output the name, a colon and space, and then the phone number on a single line. If there is no match, it should output the name, a colon, and the text, "No match found."

The output for the example inputs listed above would be:

```
Jeff: 458-5667
Dr. von Ronne: No match found.
Supercalifragilisticexpialidocious: No match found.
John Smith: +49 1234 5-67890
Abe: No match found.
Jeffery: No match found.
```

## Implementation

Your main program should be made up of the files: `strmatch.h`, `strmatch.c`, `phonebook.h`, `phonebook.c`, `pbe-darray.h`, `pbe-darray.c`, and `phonebook1.c`.

`strmatch()`. This program needs to be able to compare 'name' lines from the standard input with the names stored in phonebook entries. Write a function that takes two pointers to null-terminated strings and compares them. It should return 1 if they are the same (i.e., each corresponding element holds the same number of characters, and they end with a null-character in the same position) and return 0 if they are not the same.

This function should be defined in `strmatch.c` and a declaration with a prototype should be placed in `strmatch.h`.

Place a main function in `test-strmatch.c` that reads in pairs of input lines (up to 80-characters each) and uses `strmatch()` to determine if they are same. If they are same, the program should print a line that says "\tSAME!", and if they are not the same it should print out a line that says "\tdifferent". This is a 'unit test' driver that tests the `strmatch` in isolation, so that you can test and debug it before using it to build a bigger program.

Create a Makefile with rules to compile `strmatch.c` and `test-strmatch.c` and link their object files together into the executable `test-strmatch`. Run `test-strmatch` to unit test your code.

*struct pentry.* Put a declaration of a structure with the tag `pentry` in the file `phonebook.h`. Each instance of the `struct pentry` will represent a phone book entry. The structure should have field `name` (type: `char *`) containing a pointer to a string holding the name for the entry and a field `number` containing a pointer to a string holding the phone number for the entry.

For debugging purposes, also write a function `print_entry()` in `phonebook.c` and a `main()` function in `test-phonebook.c`. The `print_entry()` function should take a pointer to a `pentry` structure and print the structure's name and phone number's on a single line separated by a colon and space. The `main` function in the file `test-phonebook.c` should create a couple of phone book entries and stores some dynamically allocated strings containing names and phone numbers in them and then print those entries out.

Create a Makefile with rules to compile `phonebook.c` and `test-phonebook.c` and link their object files together into the executable `test-phonebook`. Run `test-phonebook` to unit test your code.

*Dynamic Array List of Phonebook Entries.* Implement a dynamic array list of `pentry` structures (call it `struct pbe_darray`) that can be used to store an arbitrary number of phone book entries and access them by index number. This dynamic array list structure should be implemented using C99's flexible array member feature. You should implement functions, for initializing the dynamic array list, appending a new entry to the dynamic array list, getting a particular entry from the dynamic array list, and freeing up the memory used by the list (including the name and number strings). Put the definitions in `pbe-darray.c` and the 'public' declarations in `pbe-darray.h`.

An example program using a dynamic array list of characters has been provided in the examples directory of your subversion repository. You may adapt the dynamic array list implementation found in the files `examples/dynamic-reverse/darray.h` and `examples/dynamic-reverse/darray.c` for use with the `pentry` structure instead of `char`. The instructor will also place some additional notes explaining the implementation in `course-notes/dyn-array-list.pdf`.

For debugging purposes, write a function `print_phonebook()` in `phonebook.c`, and a `main()` function in `test-darray.c`. The `print_phonebook()` function should take a pointer to a `pbe_darray`

structure and print out all of the `pentry`'s in the dynamic array list. The main function should initialize a `pbe_darray`, append several `pentry` structures (with dynamically allocated name and number strings), print them out using `print_phonebook()`, and free the memory used by the dynamic array list. The `pbe_darray` should be initialized with an initial capacity of small enough number to make sure that it gets to be enlarged at least once as the main function appends entries.

Create a Makefile with rules to compile `phonebook.c` and `test-darray.c` and link the object files together into the executable `Run test-darray` to unit test your code. Be sure to run `test-darray` under `valgrind` to increase the chances of detecting memory allocation/pointer problems that cause undefined behavior.

`read_phonelist()`. Implement a function `read_phonelist()` to read in pairs of names and phone (formatted as described above) from a file stream and append them to a dynamic array list of phone book entries. This function should be declared in `phonebook.h` and defined in `phonebook.c`.

The function `read_phonelist()` should take a `FILE *` parameter, which specifies the file stream from which it should read, and a pointer to a pointer to a `pbe_darray` structure, in which it will store the data read from the file. Each line from that file should be copied into a dynamically allocated string just large enough to hold that line (include `'\0'`) without any wasted space. The strings from each pair of lines should be put into `pentry` structure appended to the `pbe_array`.

Create a sample input data file, and extend the main function in `test-phonebook.c` to exercise `read_phonelist()` on that file, printing out the result. Modify the Makefile to be able to link `test-phonebook.o` with `phonebook.o` and `darray.o` to produce `test-phonebook`. Run `test-phonebook` to unit test your code.

`pbe_lookup()`. Implement a `lookup()` function that takes a string pointer to a name and a pointer to the `pbe_darray` and finds the first `pentry` in the `pbe_darray` with a matching name. If a match is found, it should return a pointer to that `pentry`, otherwise it should return the `NULL` pointer.

The `pbe_lookup()` function should use the `strmatch()` function and `pbe-darray.c` functions to perform its tasks. The definition should be in `phonebook.c` and declaration in `phonebook.h`.

Extend the main function in `test-phonebook.c` to also exercise a successful and unsuccessful search with `pbe_lookup()`. Run `test-phonebook` to test your new code.

`main()`. Write a main function in the file `phonebook1.c` that will use the functions described above to carry out the programs tasks. Specifically, it should read in the file specified on the command-line in, pulling out the 'name' and 'number' strings from the file (as described above), dynamically allocating space for those strings, and placing them into a dynamic array list of `pentry` structures. (To make testing easier, use an **initial dynamic array list capacity of 5**.) The main function should, then, read in a list of names (one per line) from the standard input. For each name read from the standard input, the program should determine if there is an exact match in the dynamic array list. If there is a match it should print out the name, followed by the phone number. If there is not a match, it should print out the name followed by the string "No match found."

Add a Makefile rule to compile `phonebook1.c` and a rule to link `strmatch.o`, `phonebook.o`, `pbe-darray.o`, and `phonebook1.o` into an executable called `phonebook1`.

Test your program thoroughly using `valgrind` and a variety of valid and invalid inputs.

## Testing/Debugging

In implementing this assignment, there are lots of opportunities to introduce bugs that only show up only rarely. In general, some of the reasons for this can be (1) that the code branch containing the bug may only be executed on certain inputs, (2) that the error may only be observable for certain inputs (often related to ‘boundary conditions,’ such as the size of the input related to the size of internal buffers), or (3) that the error may result in undefined/nondeterministic behavior which happens to be what the programmer expects most of the time. Problems (1) and (2) become more problematic as programs get more complex. A thorough regimen of unit testing is helpful in addressing these problems at least for faults that are entirely contained in a single code unit. Problem (3) can result from memory allocation / pointer bugs, and so may be more of an issue with the programs you write in this class than with programs you have written previously.

Here are some techniques/tools for help you find faults in your program:

**testing boundary conditions** Software failures are often the result of getting a test ‘off-by-one’ (e.g., using `<` when `<=` was meant). For this reason it is often useful to test programs just on both sides of the ‘boundaries’ where the programs behavior should change. (These boundaries can be found by looking at the specification of what the code should do. A related technique is to look at the if statements in the code and make sure, you’ve tested inputs which execute both the ‘then’ and the ‘else’ parts of as many if statements as possible.) For example, in this assignment, lines longer than 80 characters should be treated as if they were truncated to 80 characters, so it would make sense to test the program with lines, which are 79 characters, 80 characters, and 81 characters along with lines that are much shorter or much longer.

**unit testing** As software systems get bigger and more complex, the possible execution paths grows super-linearly and it becomes infeasible to test all of the combinations. Well-designed software is divided into modules that have a well-defined interface (this is one of the primary purposes of classes in OO languages). Testing that each module obeys its interface is usually more tractable than testing the whole program. Testing of a single module (or unit) is called unit testing and can be more effective at finding the bugs within that unit than whole program testing. (On the other hand, another source of bugs is the complex interactions between modules, and to find those you need to also test the whole program (systems testing), subsystems within the program, or groups of modules together (integration testing). The ‘test’ programs you implemented above are examples of unit and integration testing.

**MALLOC\_CHECK\_** The GNU libc memory allocation functions understand the environment variable `MALLOC_CHECK_`. If this is set to 3, `malloc()` and friends will do some extra checks each time they are run (e.g., make sure you don’t free the same pointer twice) and aborts the program execution with an error message when such checks fails. This slows the program down, but helps you catch and fix more bugs.

If you are using the `tcsh` shell (which you will on the department machines), you can set the environment variable with the command:

```
setenv MALLOC_CHECK_ 3
```

You should do this before you are executing your program. The checking can be turned off by replacing 3 with 0.

**valgrind** A more extensive checker is valgrind. It uses virtual machine technology to dynamically check a range of pointer/allocation bugs (uninitialized variables, buffer overruns due to bad pointer arithmetic, double frees, etc.) during a program's execution. To use valgrind, you simply prefix the word valgrind to the program execution command:

```
valgrind ./phonebook1 phonenumber.txt
```

Valgrind prints out a lot of stuff, even when your program is working correctly. It is worth spending the time to understand what those things mean, but it gets a little old looking at it all each time the program runs. If you give valgrind the `-q` switch before the program-to-be-tested's name, valgrind will only print out messages for errors. Use of valgrind in testing your program is highly recommended. **If your submitted program has bugs that are easily detected using valgrind but not reported in your README, additional points may be deducted.**

## Submission

Please create a README file divided into three sections: NOTES TO GRADER, PROGRAM VALIDATION, and CERTIFICATION.

1. Under NOTES TO GRADER, add any notes, you wish the TA/instructor to consider while grading.  
Please document any known problems with your program. If gcc produces any warning or error messages (with the switches: `-std=c99 -pedantic -Wall -Wextra -Wstrict-prototypes -Wno-unused-parameter`), or if valgrind reports any problem, please list them, describe what you think they mean, and say whether you think it indicates a problem in your code or if gcc/valgrind is just being 'too picky.'  
Please, also note approximately how many hours you spent to complete the assignment. This information will not effect your grade, but will be used for future course planning.
2. Under PROGRAM VALIDATION, please describe what you did to test your program or otherwise check that it is behaving correctly. In particular: What inputs did you try? How did you determine if the outputs were correct? Did you try any invalid inputs and make sure that your program doesn't crash? Have you tried input lines that are 79, 80, 81, and more than 81 characters long? How about invalid command-line arguments (e.g., no filename, a filename that doesn't exist, more than one parameter)? How about input files with an odd number of lines? Did you use valgrind when running these tests?
3. Under CERTIFICATION, please answer the following two questions in your README file:
  - (a) Did you use any code/materials other than those provided by or referenced by the instructor or TA in completing this assignment? If so, please cite the source (including its URL if it is online) and the nature and extent of your use of that material.  
If such material is prohibited by the directions for this assignment, points may be deducted, but as long as such use is completely disclosed in the README file (including source, nature, and extent of use), it will not be considered to constitute scholastic dishonesty.

- (b) Did you work with anyone or get help in completing this assignment from anybody other than the instructor or TA? (This includes mentors and tutors.) If so, please describe the nature and extent of the help received.

If such help or collaboration exceeds what is allowed for this assignment according to these directions, points may be deducted, but as long as the help or collaboration is completely disclosed in the README file, it will not be considered to constitute scholastic dishonesty.

Below your answer to those two questions, include the following statement:

I certify that except as noted above, the work I submit for this assignment has been completed solely by me without any outside help and without looking at any code for solving this problem other than what is contained in this document or included in other material/links provided by the instructor or TA.

Below this statement, please write your name and date.

Use svn to commit your final submission by 5pm on the due date.

1. Make sure that you have all of the required files and they've all been added to svn. The the assgn4 directory in the subversion repository is expected to include: assignment5.pdf, README, Makefile, strmatch.h, strmatch.c, test-strmatch.c, phonebook.h, phonebook.c, test-phonebook.c, pbe-darray.h, pbe-darray.c, test-darray.c and phonebook1.c. It is also a good idea to view your repository with your web browser to make sure all of the files are there and contain the most up-to-date contents.
2. You can continue to make changes, and re-commit as often as you want. The last version committed before 5pm on the due date will be graded.

*Permissible Collaboration and Resources.* This assignment is to be completed *individually*. You are allowed to discuss general strategies for solving assignments with fellow students or other individuals, but it is no longer a 'general strategy' if the discussion gets to the level of detail of what would be done in actual lines of code found in your programs. In addition, discussions of 'general strategy' should not be taking place while either party is editing their source code. (It is, however, perfectly acceptable to discuss C language constructs and concrete examples of them which are not taken directly from anyone's solution to an assignment.) In addition, you may seek more specific help from mentors and lab tutors in trying to understand why their code doesn't work. Furthermore, you may also seek help and guidance of any kind from the TA and instructor.