Module 110 Project: Stack Variations
CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
May 3, 2016

The Pairs

- Erhaan Ahmad and Lilly Webster
- Sanjay Sudhir and Gemma Nash
- Lex Martin and Faizaan Ali
- Tanner Tufto and Zachary Susag
- Dennis Chan and Cory McCartan
- Jae Hong Shin and Eli Salm
- Ying Zhang and Tyler Williams
- Mattori Birnbaum and Anna Blinderman
- Sophie Gaschott and Saung Thuya
- Yuyin Sun and Jong Hoon Bae
- Ella Nicolson and Josh Lavin

Colin Greenman will take the place of anyone who is absent.

The Assignment

As you develop your programming skills, you should become as comfortable reading, amending, and adapting someone else’s code as you are writing your own code from scratch. In this project, we’ll read, amend, and adapt some code originally written to compare array-based implementations of stacks of strings as an abstract data type.

The code, which you can find in the /home/stone/c/code/module-110 directory on MathLAN, includes four such implementations (stack-lab-1.c, stack-lab-2.c, stack-lab-3.c, stack-lab-4.c) and three applications that use those implementations (stack-lab.c, stack-project.c, and stack-project-compare.c).

Your goal is to study the code in these seven files and to revise and improve it until all three applications run with all four implementations, improving the programming style as you go, and then to compare the efficiency and memory use of the four implementations in the context of the applications (or simple variations of them).

This time, instead of turning in code, you are to submit a prose report describing the changes that you made, the obstacles you encountered and the solutions you worked out for removing or avoiding those obstacles, the strengths and weaknesses of your final version of the code, and the results of your comparisons of the four implementations of stacks of strings.

The Implementations

Your first step should be to read through the files and get an idea of what they are about. You may want to take some notes recording things that don’t look right or are simply confusing, perhaps in the form of questions that you hope to be able to answer later either by careful reading of the code or by experimentation.

The implementation files for this project present four different approaches to implementing a stack, all based on an underlying array structure. In particular, each approach uses an array of fixed size to store strings on a stack.

In stack-lab-1.c and stack-lab-3.c, the elements of the array are of type char *, and storage for each string that enters the queue is either provided by the caller (stack-lab-1.c) or dynamically allocated by the push operation (stack-lab-3.c). In stack-lab-2.c and stack-lab-4.c, on the other hand, the elements of the array are themselves
fixed-size arrays of characters, allocated as part of the stack itself, and the push operation copies the given string into one of those character arrays.

The stack-lab-2.c and stack-lab-4.c implementations are distinguished by their handling of the pop operations. In stack-lab-2.c, pop dynamically allocates storage for the string that it returns, while in stack-lab-4.c pop returns the address of character 0 in the fixed-size array occupied by the string being popped. In stack-lab-1.c and stack-lab-3.c, on the other hand, pop simply returns the pointer it finds in the relevant array location.

To summarize:

In stack-lab-1.c (the “passing pointers” implementation), the stack contains an array of pointers, exactly as pushed; returned strings are those same pointers. The caller is expected to provide the storage that the strings occupy.

In stack-lab-2.c (the “copying strings” implementation), the stack contains an array of fixed-length character arrays. Strings are copied into these arrays when they are pushed and copied out to dynamically allocated blocks of storage when they are popped.

In stack-lab-3.c (the “allocating strings” implementation), the stack contains an array of pointers, pointing to dynamically allocated blocks of storage into which the strings are copied as they are pushed; returned strings are these same pointers.

In stack-lab-4.c (the “pointers-to-copies” implementation), the stack again contains array of fixed-length character arrays, and strings are copied into these arrays when they are pushed. The pop and top functions return the addresses of these copies within the stack.

The Applications

These are three test programs that seem to have been developed more or less independently, perhaps by three different authors trying to exercise slightly different features of the interface. The stack-lab.c program exercises all of the functions provided and uses three different stacks, demonstrating that they can be operated on independently. The other two files use only initialize, push, and pop. The stack-project.c program seems to have been intended to detect possible errors in storage management, while stack-project-compare.c provides the basis for a test of running-time.

Compiling and Running the Programs

After reading the files carefully, your next step should be to try to compile and run them. This is complicated by the unorthodox module structure that the authors have used.

In a collection of programs that profess to be about stacks as an abstract data type, one would expect to find a header file, perhaps called stacks.h, that specifies the application programming interface to this ADT. That’s not the case here. Instead, each of the .c files containing one of the stack implementations incorporates the common interface, and the application programs get access to them by including the entire .c file.

This is a kludge. Under this arrangement, it is useless to compile any of the stack implementations separately, and changing an application program from one stack implementation to another requires editing the #include line and recompiling the application program. (In a properly designed implementation, it would only be necessary to link different .o files to make a new executable; nothing would have to be re-edited or recompiled.)

Eventually you may want to rewrite the code to provide a stacks.h file and separate compilation. On the first run, however, you should work as the authors intended,
editing and compiling each of the applications programs four times, once for each stack implementation.

Again, make notes about any warning messages and error messages that you receive during compilation—you’ll want to follow up on them to figure out what is producing them and how to avoid them. Similarly, when you run the resulting executables, examine the output carefully and record any deviations from the output you expected to see based on your reading of the source code.

Debugging

As you investigate the errors that you find, it may help to write down the questions to which you need answers, your hypotheses about the causes of the errors, the tests you conduct and the evidence you collect to support or refute these hypotheses, and the changes that you make in order to get the code working.

The gdb debugger can help with these investigations.

Amending the Code

Once you have versions of the code that work correctly, you may want to experiment with ways of simplifying and clarifying it, without losing correctness. Go through the code again, recording possible stylistic improvements. Choose some that seem possible and carry them out, retesting afterward to make sure that everything still works.

Timing

If you add the word time at the beginning of any command that you give to a shell, the shell will not only run the command but print a short summary afterward indicating how long it took to execute the program, in the following format:

```
$ time ./stack-project-compare
real  0m0.036s
user  0m0.018s
sys   0m0.015s
```

For stack-project-compare as written, the times will probably be too short to measure accurately, so you should consider increasing the values of the constants MaxStack and StringLength.

You may find that there is some variation in the timing, due to inconsistencies in the behavior of our local-area network and in the other processes that are running on your workstation. You may want to run the command several times and take the mean of the timings. Don’t forget to record your observations.

To measure the memory use of these programs, you can ask valgrind to keep track of the allocations. Since none of the dynamically allocated memory is ever freed, valgrind will report it all as leaking in its summary. You may also want to keep track of the size of the statically allocated stacks in the application programs as you switch from one implementation of stacks to another.

Submitting the Reports

Your reports will be due at the beginning of class on Friday, May 6. I prefer to receive them in hard copy, typed and printed, double-spaced, with generous margins (at least an inch and a quarter on each side). Make sure that both (or all) of the names of your team members are on the paper you submit.