Lab 3: Pointers and arrays

Lab sessions Tue Oct 17 to Thu Oct 19

Lab written by Julie Zelenski

Learning goals

During this lab, you will:

1. investigate how arrays and pointers work in C
2. use gdb and valgrind to debug memory errors
3. learn a little about C file I/O

Find an open computer and somebody new to sit with. Introduce yourself and talk about the awesome touchdown that bunny scored at Saturday's game against Oregon.

Get started

Clone the repo by using the command below to create a lab3 directory containing the project files.

```
git clone /afs/ir/class/cs107/repos/lab3/shared lab3
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab3).

Lab exercises

1) Code study

Pointers and arrays

The file `code.c` contains a nonsense C program for you to use to observe the mechanics of arrays and pointers.

1. Build the program, start it under gdb, and set a breakpoint at `main`. When you hit the breakpoint, use `info locals` to see the state of the uninitialized stack variables. Step through the initialization statements and use `info locals` again.
2. The expressions below all refer to the local variable `arr`. First try to figure out what the result of the expression should be, then evaluate in gdb to confirm that your understanding is correct.

```
(gdb) p *arr
(gdb) p arr[1]
(gdb) p &arr[1]
(gdb) p arr + 2
(gdb) p &arr[3] - &arr[1]

(gdb) p sizeof(arr)
(gdb) p arr = arr + 1
```

3. The main function initializes `ptr` to `arr`. If you repeat the above expressions with `ptr` substituted for `arr`, most (but not all) have the same result. The first five evaluate identically, but the last two produce different results for `ptr` than `arr`. The size is not the same and you can assign to `ptr` but not `arr`. A stack array and a pointer to it are almost interchangeable, but not entirely. Can you explain those subtle differences?
4. Use the gdb `step` command to single-step from `main` into the call to the `binky`. Once inside binky, use `info args` to see values of the two parameters. Print any expression you can think of on `a` and `b` and they will evaluate to the same result. This includes the last two expressions from above: `sizeof` reports the same size for `a` and `b` and assignment is permissible for either. What happens in parameter passing to make this so? Try **drawing a picture** of the state of memory to shed light on the matter.
5. Set a breakpoint on `change_char` and continue until this breakpoint is hit. When stopped in gdb, use `info args`. The arguments shown are from the "frame of reference" which corresponds to the function currently executing.
6. You can select the frame of reference with the gdb `frame` command. Use `backtrace` to show the sequence of function calls that led to where the code is currently executing. Frames are numbered starting from 0 for the innermost frame. Try the command `frame`

`1` to select the frame outside `change_char` and and then use `info locals` to see the state from `winky`. The gdb command `up` is shorthand for selecting the frame that is one higher from current.
7. Step through `change_char` and examine the state before and after each line. Use `info args` to show inner frame and `up` and `info locals` to show what's happening in outer frame. Careful observe the effect of each assignment statement.
8. Step through the call to `change_ptr` and make the same observations. Which of the assignment statements had a persistent effect in `winky` and which did not? Can you explain why?

If you don't understand or can't explain the results you observe, stop here and sort it out with your partner or your lab leader. Having a solid model of what is going on under the hood is an important step toward understanding the commonalities and subtle differences between arrays and pointers.

File I/O

Dealing with input/output is one of the less exciting tasks to handle as a programmer. Every programming language has an I/O facility, they are all different and quirky in their own ways, and they all have a vast set of features that you should not bother to try to learn in advance, just look up on a need-to-know basis.

Peruse our guide to the C standard library (/class/cs107/guide/stdlib.html) for an overview of some of the common C I/O functions. The `count_input` and `main` functions in the `mywc.c` file show idiomatic code to open a file and read text line-by-line. Discuss these questions with your partner:

- How can you detect that a file cannot be opened?
- What are the possible options for the `mode` argument to `fopen`?
- What is the consequence of failing to `fclose` a file?

Standard input

The unix command `head filename` prints the first 10 lines of the named file. Its man page says that if no filename argument is given, the command will read from standard input, which means it will process text entered by the user by typing at the terminal. Almost every unix utility that operates on files (e.g. `grep` `cat` `wc` `less` `head` `sort` and more) conforms to this same interface. Reading from standard input is an incredibly useful feature when stringing together unix commands into pipelines (https://en.wikipedia.org/wiki/Pipeline_%28Unix%29) and can also be handy for running a quick test without the overhead of creating a file.

Try this out for yourself:

1. View the contents of the nations (http://tmbw.net/wiki/Lyrics:Alphabet_Of_Nations) file in the samples subdirectory using the command `cat samples/nations`.
2. Use that file as the input to sort: `sort samples/nations`.
3. Execute `sort` without a filename argument. The program will wait for you to type input.
4. Type a few lines of text: `red` `blue` `green` ... with each word on its own line.
5. After entering a few lines, press `control-d` on your keyboard on a line by itself. This signals the end of the input.
6. It should now sort the lines you entered and print them.

That is how interact with standard input as a user; so how does a program implement this behavior? Take a look at the `main` function in the `mywc.c` file to learn how. Discuss these questions with your partner:

- How does the program detect whether there was a filename argument?
- How does it switch to reading standard input when there is no filename argument?
- Are the operations that read from a file also used to read standard input or is different code required for each?

2) Tools: tracking memory errors

We have more troubled code to use as practice with using the tools gdb and Valgrind. The `buggy.c` program has four planted memory errors, one that misuses stack memory and three that misuse heap memory.

Stack protector

1. Consider error #1 from the `buggy.c` program. When the program is invoked as `./buggy 1 name`, it will copy name into a stack array declared with space for 6 characters. If name is too long, this code will write past the end of the array into the neighboring space. This kind of error is called a *buffer overflow* (because the write overflows past the end of a buffer)

and results in *stack smashing* (destroys data stored on stack next to the buffer). Let's find out what is the observed consequence of stack smashing.

2. Run `./buggy 1 eliza` to see that the program runs correctly when the name fits. Now try the longer name `./buggy 1 hamilton`. Surprisingly, this also seems to work, apparently getting lucky because the overrun is small. Push the program a wee bit further `./buggy 1 alexander` and you'll be rewarded with a show-stopping `*** stack smashing detected ***`.
3. Run this last case again under gdb. When program hits the error, use `backtrace` to see what reported it. It looks like something in the library intervened at the buffer overflow. The detection of a stack overrun is a safety feature called "stack-protector (http://wiki.osdev.org/Stack_Smashing_Protector)" provided by the gcc compiler.
4. Stack protector is controlled by a compiler flag. Let's disable it and see what happens without it. Open the Makefile in your editor and change the CFLAGS entry `-fstack-protector` to `-fno-stack-protector`. Use `make clean` to remove the previous build and `make` to rebuild.
5. Now run `./buggy 1 linmanuelmiranda`. No more helpful report about stack smashing, instead just your nemesis `Segmentation fault (core dumped)`.
6. Run this last case again under gdb. At the point of the crash, use `backtrace`. Woah, what the heck has happened to the stack? One critical piece of data stored on the stack is the sequence of active function calls. Without the vigilance of stack protector, the buffer overflow went unchecked and destroyed that vital information, and gdb cannot even tell you how we got here. Later this quarter, we will explore how the stack housekeeping is managed and the vulnerability of that information. For now, we will be glad to have the help of stack protector to detect and report stack smashing.

Valgrind

The remaining memory errors planted in `buggy.c` are fodder for more practice with Valgrind. Last week's lab introduced you to Valgrind and this tool will be increasingly essential as we advance to writing C code with heavy use of pointers and dynamic memory.

1. Consider error #2 from `buggy.c`. The erroneous code is similar to buggy error #1, it copies the name argument to a space allocated to hold only 6 characters. Instead of a long name overrunning a stack-declared array as before, this error overruns a heap-allocated array.
2. Run `./buggy 2 eliza` and try again with longer names `hamilton` or `alexander`. Hmmm, it seems to work, perhaps a heap overrun has a easier time "getting lucky" than a stack overrun? Try again with longer names and eventually you will get the crash you deserve. How long a name did it take?
3. Run under valgrind: `valgrind ./buggy 2 philip` and read the report to see how the error is reported. Thankfully Valgrind will not let anything slide --even just one byte too many is swiftly rejected!
4. The error cases #3 and #4 are due to mishandling freed heap memory. First review the code to see what these errors are. Then run `buggy 3` and `buggy 4` at the shell and again under Valgrind. Read how each error is reported by Valgrind and take note of the terminology that the Valgrind report uses for each.

Be forewarned that memory errors can be very elusive. A program might crash immediately when the error is made, but the more insidious errors silently destroy something that only shows up much later making it hard to connect the observed problem with the original cause. The most frustrating errors are those that "get lucky" and cause no observable trouble at all, but lie in wait to surprise you at the most inopportune time. (e.g. when we are grading your work:-) Cultivate the habit of using Valgrind early and often in your development to detect and eradicate memory errors before they strike.

3) Write, test, debug, repeat

The `mywc.c` file implements a `wc`-like program. We give you a mostly implemented but buggy implementation and your task is to test and debug the program into a fully functional state.

1. Read `man wc` to get info on the standard Unix utility. The `mywc` program is a slightly tweaked version that only counts lines and characters and additionally reports the longest line.
2. Review the starter code and verify that you understand how it operates. The given code does correctly count the number of lines and characters, but cannot reliably print the longest line.
3. Run sanitycheck. The code passes the first two cases, but fails the third. Run that third case under gdb, single-stepping and printing state to figure out what's going wrong. Discuss with your partner what you observe. Identify the memory misunderstanding of the code's original author.

4. Add a call to the function `strdup` (`man strdup`) to fix the problem. Don't attempt to free anything, just let it leak memory for now. Run under sanitycheck to see that your fix now allows the program to pass all tests.
5. Run under valgrind to see how a leak is reported. At the end of the valgrind report will be a recommendation to re-run with additional flags. Re-run valgrind, adding its suggested flags. Do you see how this information helps to identify the origin of the leak?
6. Add in the appropriate free call(s) and re-test under sanitycheck and Valgrind. It may take a few iterations for you to work out what calls are needed and where. As a rule, there needs to be a one-to-one correspondence between each call to malloc and each call to free.

Check off with TA

At the end of the lab period, submit the checkoff form and ask your lab TA to approve your submission. Take stock of your understanding with our selfcheck (/class/cs107/selfcheck.html#lab3).