Lab 2: C-strings

Lab sessions Tue Oct 10 to Thu Oct 12

Lab written by Julie Zelenski

Learning goals

During this lab, you will:

1. read and analyze C code that operates on chars and C-strings
2. use Valgrind to detect and debug memory errors
3. write some of your own code to wrangle C-strings

First things first! Find an open computer to share with a partner. Introduce yourself and tell them about your favorite music to listen to while coding.

Get started

Clone the lab repo by using the command below. This command creates a lab2 directory containing the project files.

```
git clone /afs/ir/class/cs107/repos/lab2/shared lab2
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab2).

Lab exercises

1) Code study

The code excerpts for this week's lab are taken from the standard C library. By studying this code, we can learn how the library functions are implemented, as well as become better informed about how to properly use these functions as a client and what consequences to expect when we don't.

ASCII and ctype

`<ctype.h>` offers small utility functions such as `isdigit` and `tolower` that operate on single characters. The ctype function `isxdigit` below reports whether a given character is a valid hex digit.

```
int isxdigit(int c)
{
    return isdigit(c) || ((unsigned)c|32)-'a' < 6;
}
```

Here are a few questions to work out with your partner. If you need an ASCII refresher, use `man ascii`.

1. It seems the first test checks for a digit character, so the second test must be in charge of validating hex letters, but its expression is rather mystifying. What is the purpose of the bitwise OR with 32 and the character subtraction?
2. The cast to `unsigned` is deliberate and necessary. Get into gdb and print the expression where c is `'#'` both with and without the cast, e.g.:

   ```
   (gdb) p ((unsigned)'#'|32)-'a' < 6
   ```

   You'll see that the result is different without the cast. Why?
3. Use `man string` to see the list of string functions in the standard library. You could reimplement `ixdigit` as a single call to one of them. Describe how.

strcpy and strncpy

Below are simple implementations for `strcpy` and its cousin `strncpy`. For variety, I chose a `strcpy` that operates via pointer arithmetic as a contrast to the `strncpy` that uses array-indexing.

```
char *strcpy(char *dst, const char *src)
{
    char *result = dst;
    while ((*dst++ = *src++)) ;
    return result;
}

                                  // taken from the strncpy man page
char *strncpy(char *dest, const char *src, size_t n)
{
    size_t i;

    for (i = 0; i < n && src[i] != '\0'; i++)
         dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

Points to ponder:

1. Dissect the expression used as while loop test in `strcpy`. That single statement copies a char, advances the two pointers, and tests the stopping condition all in one! How does it work?
2. What happens if you call `strcpy` passing a `src` string that is missing its null terminator? What if `src` is NULL or an invalid address?
3. Neither function checks if `dst` is large enough to hold a copy of the string. Such protection against user error would be very welcome, but sadly is not possible in C. Why not? So what *is* going to happen if `dst` is too small?
4. `strcpy` appends always and exactly one null-terminator to the characters copied to `dst`. How many null-terminators are written by `strncpy`? (Be careful!)

Aside: In the days of yore, pointer arithmetic had a performance advantage over array indexing, but smart modern compilers make this largely moot. A lot of old code and old programmers haven't yet gotten that memo. Pointer arithmetic is especially rife in code that wrangles C-strings where it is perhaps tolerable, but for arrays of type other than char, array indexing is almost always going to be more readable.

2) Tools interlude

Each lab will devote some time to hands-on practice with the tools. Last week, it was sanitycheck and gdb. This week, we'll do a little more with gdb and introduce Valgrind.

gdb x command

1. Load the `code` program under the debugger, set a breakpoint at `main` and run the program.
2. When you hit the breakpoint, try out the gdb `x` command which is used to examine raw memory:

   ```
   (gdb) x/8bc buf
   ```

   `x` shows the contents of memory starting at a given address. In the command above, the modifiers `/8bc` tell gdb to print 8 bytes interpreting each byte as character format (ascii). Use `help x` to read about other available modifiers. What modifiers would print 3 bytes in hex format instead?

3. Use `next` to step execution through the strcpy calls and use `x` to show the contents after each line to see how buf is changing. Verify that what you observe corroborates with your understanding of the behavior of `strcpy` and `strncpy`.

Using Valgrind

Now that we are moving on to more significant use of pointers, our programs become more at risk for memory errors. These can be difficult bugs to track down and Valgrind is a supremely helpful tool to have in your arsenal. However, it takes some practice to learn how to interpret a Valgrind report. If you haven't already, review the our guide to valgrind (/class/cs107/guide/valgrind.html) and let's try it out now.

- Read over the code in `buggy.c` to see its 3 planted memory errors. Consider just error #1 to start.

- Run `./buggy 1` and you should be rewarded with a `Segmentation fault (core dumped)`. A segfault results from an attempt to read/write an inaccessible/invalid address. Ok, so now you know your program as a memory error but not much else.
- Run buggy under gdb. You observe the same segfault, but now can use the gdb `backtrace` command to learn where execution was at the time of the crash. That's helpful!
- Exit gdb and run under Valgrind: `valgrind ./buggy 1`. The Valgrind report gives more detail including type of memory error, execution callstack, and the value of the invalid address. Furthermore, Valgrind also detected an earlier problem that preceded the seg fault -- an uninitialized value of size 8 (sizeof pointer) right before the invalid read of size 1 (char). This is the very clue need to lead you to the right place to fix!
- Repeat steps 1-4 for `buggy 2` and `buggy 3` and dig into the Valgrind report for each. Note the terminology that Valgrind uses for different kinds of errors. Your goal is to learn how to relate the error as reported by Valgrind back to the root cause.

All three buggy programs have a memory error, but the visibility of the error differs. `buggy 1` crashes with a seg fault: every time, every run. `buggy 2` has a visible flaw in that its output is garbage/unpredictable (did you note the change in running with Valgrind and without?), but it does not crash. `buggy 3` appears work correctly, the output looks just fine and program successfully runs to completion. A memory error may cause no *visible* harm, but that doesn't mean the program is correct, it just "got lucky" this time. By keeping Valgrind at your side, you get a vigilant partner that can detect these errors that are lying in wait, even before they produce an observable effect.

We recommend that you run Valgrind early and often during your development cycle. Focus on the first error reported (it often cascades from there), find, and fix, and repeat any remaining errors. Don't move on until you get a clean report from Valgrind. Note that memory *leaks* don't demand the immediate attention that *errors* do. Leaks can (and should) be safely ignored until the final phase of polishing a working program. When grading assignments, we will run your submission under Valgrind and expect to see clean bill of health, so be sure you have checked it yourself before calling it done.

3) Write, test, debug, repeat

In this part of the lab, you will write a version of the `printenv` program, used to show the "environment variables" that pertain to your terminal session. The environment is a list of key-value pairs that provide information about the context for your terminal session and configure the way processes behave. Type `echo $USER $HOST $HOME $SHELL` to a few of your environment variables. You have already used the USER environment variable when you cloned your assignments; the USER environment variable is set to your SUNet ID when you log into myth. Other environment variables include PATH (where the system looks for programs to run), HOME (path to your home directory),and SHELL (what command line interpreter you are using; most people should be using `/bin/bash` in cs107).

You will write your own `myprintenv` and will use `env` to test it. First, practice using the standard commands:

1. `printenv` will show your environment variables, `env` allows you to change them. Read the man pages for `printenv` and `env`.
2. Run `printenv` with no arguments and skim its output. Then try `printenv USER` and `printenv SHELL`. What is the output from a bad request `printenv BOGUS`?
3. Run `printenv`, then run `env BINKY=1 WINKY=2 printenv`. What changes between the two? Run `printenv` again with no arguments to determine whether changes to the environment were transient or permanent.

Now go on to coding the implementation of `myprintenv`.

1. Review the code in `myprintenv.c`. The program is mostly complete, save the implementation of the `get_env_value` helper function.. As you review the given code, be sure to take note of the 3-argument variant of the `main` function it uses to access the environment. Read the comment in `myprintenv.c` for more info.
2. Implement the `get_env_value` function to search the environment for a variable by name. Take advantage of the handy `<string.h>` library functions to help with this task.
3. Use sanitycheck (/class/cs107/sanitycheck.html) to test your work. Edit the custom_tests file for additional test coverage.

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#lab2).