Lab #2: Introduction to Pointers

Overview

This lab is meant to serve as an introduction to pointers.

To begin, download the `lab2.tar.gz` file from the course website and extract the contents directory storing your coursework.

Part 1

Write and test a `swap` function that takes two integer pointers and swaps the values at the locations referenced.

Part 2

Copy your files from Part 4 of Lab 1. Modify your implementations (but not the tests) of the two functions such that iteration over the string arguments is done with pointers and dereferencing instead of indexing.

Part 3: `gdb`

This `gdb` exercise explores the examination of pointers. Build the provided program (be sure that the program builds for debugging, i.e., with the `-g` flag). Execute the following commands to gain some experience using gdb with pointers.

```
% gdb example
...
(gdb) b example.c:33
(gdb) r
(gdb) print exp
(gdb) print *exp
(gdb) print exp->c
(gdb) print exp->other
(gdb) print *exp->other
(gdb) print exp->other->a
(gdb) c
(gdb) list
(gdb) back                    <----- how did we get here?
(gdb) quit
```

Pay careful attention to the response following the `c` command. This is one of the simplest ways to discover where (though not necessarily why) a segmentation fault occurs. If for nothing else, use `gdb` to help you debug segmentation faults.

Part 4: More `gdb`

This part of the lab will demonstrate some additional features of the GNU debugger.

Recall that to use the debugger to its fullest extent, you will need to compile your program with the `-g` switch. Modify the given `Makefile` to add the `-g` switch to the `CFLAGS` variable and then build the program.

Start `gdb` and execute this sequence of commands. Pay particular attention to the responses from `gdb`.

```
% gdb example
(gdb) b main
(gdb) r
(gdb) watch i           <---- set a watchpoint
(gdb) c
(gdb) c
(gdb) c
(gdb) c
(gdb) c
(gdb) c
(gdb) quit
```

`gdb` supports many more features than those shown here. You can type `help` at the `gdb` prompt to get a list of the features. You might also find it useful to use `n` instead of `s` to step through statements. The difference is that `s` will step "into" a function invocation whereas `n` will step over a function invocation.

Consider the following example of a somewhat more interesting use.

```
% gdb example
(gdb) b main
(gdb) r
(gdb) print &i
$1 = (int *) 0xXXXXXXXX  <--- not actual output, keep track of the real number
(gdb) b fact
(gdb) commands 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>print *(0xXXXXXXXX)     <--- actual number from above
>end
(gdb) c
(gdb) c
   ... continue until you understand exactly what is happening
```

Part 5: Bad Code

Compile and run the example code. Ignore the warnings for now (compilers continue to improve the warnings that are given).

Trace through the program and explain the output. Is this output guaranteed? No, certainly not, there are multiple bugs. The compiler likely warned about one of the more obvious bugs, but the others are more difficult for the compiler to find (slightly more subtle) and more representative of the sort of mistake one might make when creating dynamic data structures (such as a linked list).

Be sure that you can explain why these are bugs.

Part 6: `valgrind`

`valgrind` provides, among other features, a memory check tool. This tool can help identify common memory-related errors and, as such, can help you identify bugs in your programs.

Read through the Valgrind Quick Start (it is very short) linked from the course website. Compile and execute the example shown in the Quick Start.

Note: Some assignments will require a clean report from `valgrind` to receive full credit.

Part 7: Scripted Testing

As you develop your solution to an assignment, you are encouraged to write unit tests to verify that each function works as expected. In addition, as you near assignment completion, you will want to test that the program as a whole works (sometimes called system testing).

In the provided files there is a simple bash script that will run a series of tests. Each test redirects an input file (using <) into the subject program (for this example, `cat` is used as the program being tested) and redirects the output (using >) to another file. The output is then compared against an expected output using `diff`.

- Run the script (`./run_tests`, though you may need to change the permissions first with `chmod 700 run_tests`) and observe the output (with one intentional failed test).
- Add a test (it needn't be complicated) by adding a file to the inputs directory and a corresponding file to the expected directory.

You can (and should) modify this script to aid in your testing of future assignments.

Demonstration

After you have completed the above parts, call me over so that I can record your completion of this lab.