Raw | Blame

```
209 lines (162 sloc)   8.97 KB
```

layout	title	subtitle	summary	date	released	due
lab	Pointer Lab	Looking at Pointers!	Here, we will investigate the nature of pointers. They aren't here to hurt us. They are here to help us!	2020-01-27 21:04:31 +0200	12:00 AM Tuesday, January 28th, 2020.	11:59 PM Friday, February 7th, 2020.

Weight: 5% of your total 50% Lab Grade

Introduction

This lab will give you practice with pointers and pointer arithmetic. Pointers are a critical part of the C language. C gets out of your way and lets you mess with memory rather freely. Therefore, learning to use pointers and their nuances will aid you in exploring the memory layout of a system and how memory allocation works in the future.

The assignment contains a skeleton for some brand new programming puzzles! Each puzzle is a relatively empty function that has a comment block preceding it describing exactly what the function must do and what restrictions exist on your potential implementation.

Your task, should you choose to accept it (please do) is to complete each function.

Rules

Generally, each puzzle has some generic rules which can be relaxed if mentioned by the comment block. However, the following rules will apply to all:

- Only straight-line code (i.e., no loops or conditionals) unless otherwise stated.
- A limited number of C arithmetic and logical operators.
- No constants larger than a byte (8 bits) are allowed.
- Free free to liberally use `(` and `)` and `=` (assignment) as much as you want!
- ***You are permitted to use casts for these functions!***

You will start working with basic pointers and use them to compute the size of different data items in memory, modify the contents of an array, and complete a couple of pointer "*puzzles*" that deal with alignment and array addressing.

Downloading

Your lab materials are contained in an archive file called `pointerlab.zip`, which you can download to your Linux machine as follows:

```
wget {{ site.url }}{{site.baseurl }}/labs/02/pointerlab.zip
```

This will download the `pointerlab.zip` file to your (private) directory on the `thoth` Linux machine (make sure you are logged into this!) in which you will do your work. Then issue the command:

```
unzip pointerlab.zip
```

This will cause a number of files to be unpacked in the directory `pointerlab-handout`. Navigate to that directory and look at the files. The only file you will be modifying is `pointer.c`. The `pointer.c` file contains a skeleton for each of the programming puzzles.

What are these??

So, you will be updating the `pointer.c` file to perform the following tasks:

- **Pointer Arithmetic**: The first few functions of the file will ask you to compute the size (how much memory a single variable takes up, in bytes) of various data elements (int, double, and pointers in general.) You will accomplish this by noting that arrays of these data elements allocate contiguous space in memory. Notice the array provided and the two pointers. You may assign the pointers to the addresses of items in the array and then compare them. You may also find it easier to cast the addresses (pointers) to integer types, such as `long`. Make note of pointer arithmetic!

- **Manipulating Data Using Pointers**: The next two functions in `pointer.c` will investigate how manipulation of data passed in to functions through pointers affects those values. In `swapInts`, you will just copy the value addressed by the initial pointer into the memory addressed by the other, and vice versa. The `changeValue` function will update an element of an array using only the pointer to that array. That is, *you* are doing the work that the `[]` syntax we saw in lecture would be doing, which explores the finer points of pointer arithmetic. (no pun intended. ok fine, some pun intended.)

- **Pointers and Address Ranges**: It is often useful to know if a pointer falls within a particular range. In the next function, `withinBlock`, we will do exactly this.

  In `withinBlock`, you will determine if the addresses stored by the two pointers being passed in lie **within the same block** of 64-byte aligned memory. That is, if you divided up memory into nice clean 64-byte pieces, would both pointers be pointing to the same slice? Return a `1` if so, and a `0` otherwise.

  Note that when we say *aligned*, here, we mean that the first such block is at address 0, and every subsequent block is at an address that is a multiple of 64 bytes.

  Hint: consider what is true about the address of the first byte of one such block and every other block of that block. Now compare with the first byte from the next block.

  Here are some examples of `withinBlock`:

```
ptr1: 0x0
ptr2: 0x3F
return: 1
```

```
ptr1: 0x0
ptr2: 0x40
return: 0
```

```
ptr1: 0x3F
ptr2: 0x40
return: 0
```

```
ptr1: 0x3CE
ptr2: 0x3EF
return: 1
```

```
ptr1: 0x3CE
ptr2: 0x404
return: 0
```

- **String Traveral**: The next function, `stringLength` asks us to implement our own little `strlen` function. This means we go through the given string looking for that trusty null terminal (sentinel) value, which is a byte with the value of 0. (or a `'\0'` character literal) Recall that in C, strings do not store their length, so we are always beholden to functions like this. After you implement yours, notice how easy it would be to, as I like to say, Kool-Aid Man through memory if the sentinel isn't there!! (The Kool-Aid Man is owned by some large corporation. All rights reserved. I guess. Don't sue me!)

  The next function, `stringAppend` asks us to implement our own little `strcat` function. This means the function is given two strings. It will take the second string and copy it to the end of the first. Once again, we will be concerned about making good use of our sentinel values.

  Note: You may indeed use for or while loops here!

  Hint: It may be useful to use one of these functions within the other to make our lives easier!

- **Selection Sort**: Putting all of our new knowledge together (and re-using one of the functions we built earlier,) we will implement one of the most basic sorting algorithms: selection sort. This sort works by gradually building a sorted array while scanning said array. Basically, we find the smallest value and move it to the front, then sort the remaining list. Find the second smallest, etc, and so on. Slowly, the array becomes sorted as we move across. Half-way through the algorithm, the first half of the list is sorted properly, and the second half is not!

Essentially, you are writing the proper C code for the following pseudo-code (given an array `arr` and length `n`):

```
for i = 0 to n - 1
  minIndex = i
  for j = i + 1 to n
    if arr[j] < arr[minIndex]
      minIndex = j
    end if
  end for

  swap(arr[i], arr[minIndex])
end for
```

You **definitely** are allowed to use loops, ifs, and such on this one. And, **hint hint**, I think `swapInts` will be very useful, once you have implemented it.

Testing

We have included the following tools to help you check the correctness of your work automatically:

- `ptest.c` is a test harness for `pointer.c` and you can test your solutions like this:

```
thoth $ make
thoth $ ./ptest
```

- `dlc.py` is a Python script that will check for compliance with the coding rules. The usage is:

```
thoth $ python dlc.py
```

This will run silently unless it detects a problem, such as an illegal operator, constants usage, or non-straightline code (loops, ifs) in the `pointer.c` code. Although, I believe it will print out an empty list such as `[]` if it encountered no problem.

Just like in the prior lab, the `dlc.py` script enforces a slightly stricter form of C than the normal compiler, `gcc` . In particular, in a block (enclosed in curly braces) all your variable declarations must appear before any other type of statement (basically a C89 pedantic style). For example, it will complain about the following:

```
int foo(int x) {
  int a = x;
  a *= 3;
  int b = a; /* ERROR: Declaration not allowed here. */
}
```

Instead you must declare your variables first, like so:

```
int foo(int x) {
  int a = x;
  int b;
  a *= 3;
  b = a;
}
```

The `dlc.py` script also does not like the non-standard binary constant notation such as `0b10011010` so you will have to avoid this and use hex instead.

Submission

You should hand in your `pointer.c` solution file using Gradescope. It may be a couple of days until the assignment is in Gradescope, so if you finish very quickly, just wait a couple of days and check again. If you're using a Shell Terminal, you can use `scp` to copy your file from the remote machine to your local machine as follows:

```
scp username@thoth.cs.pitt.edu:/path/to/remote/file /path/to/local/file
```

If you're using Windows, you can download the WinSCP software and enter with host `thoth.cs.pitt.edu` and then your username and password. Once in, locate your `pointer.c` file and simply drag and drop to your Windows computer file system. There are also free GUI clients for Linux or Mac such as FileZilla (https://filezilla-project.org/).