Spring 2019

Lab 1a: Pointers in C

Assigned: Monday, April 8, 2019

Due Date: *Monday, April 15, 2019 at 11:59pm*

Overview

Learning Objectives:

- Gain familiarity with pointers and pointer arithmetic.

Pointers are a critical part of C and necessary for understanding assembly code (Lab 2-3) and memory allocation (Lab 5).

Code for this lab

Browser: Download here (lab1a.tar.gz)

Terminal: `wget https://courses.cs.washington.edu/courses/cse351/19sp/labs/lab1a.tar.gz`

Unzip: Running `tar zxvf lab1a.tar.gz` from the terminal will extract the lab files to a directory called `lab1a`.

Lab 1a Instructions

`pointer.c` contains a skeleton for some programming puzzles, along with a comment block that describes exactly what the functions must do and what restrictions there are on their implementation. Your assignment is to complete each function skeleton according to the following rules:

- only straightline code (i.e., no loops or conditionals) unless otherwise stated. Look for "Control Constructs" under `ALLOWED` in `pointer.c` comments.
- a limited number of C arithmetic and logical operators (described in `pointer.c` comments)

- no constants larger than 8 bits (i.e., 0 - 255 inclusive) are allowed
- feel free to use "(", ")", and "=" 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 addresses.

Using Pointers

This section describes the functions you will be completing in `pointer.c` found in the `lab1a` folder you downloaded. Refer to the file `pointer.c` itself for more complete details.

Pointer Arithmetic

The first three functions in `pointer.c` ask you to compute the size (how much memory a single one takes up, in bytes) of various data elements (ints, doubles, and pointers). You will accomplish this by noting that arrays of these data elements allocate contiguous space in memory so that one element follows the next.

Manipulating Data Using Pointers

The next two functions in `pointer.c` challenge you to manipulate data in new ways with your new knowledge of pointers.

The `swapInts` function in `pointer.c` asks you to swap the values that two given pointers point to, without changing the pointers themselves (i.e. they should still point to the same memory addresses).

The `changeValue` function in `pointer.c` asks you to change the value of an element of an array using only the starting address of the array. You will add the appropriate value to the pointer to create a new pointer to the data element to be modified.***You are not permitted to use [] syntax to access or change elements in the array anywhere in the `pointer.c` file.***

Pointers and Address Ranges

The next two functions in `pointer.c` ask you to determine whether pointers fall within certain address ranges, defined by aligned memory blocks or arrays.

For the first of these two functions, you will determine if the addresses stored by two pointers lie ***within the same block*** of 64-byte aligned memory. The following are some examples of parameters and returns for calls to this function.

-
  ```
  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`

For the last function you will determine if the address stored in `ptr` is pointing to a byte that makes up some part of an array element for the passed array. The byte does not need to be the first byte of the array element that it is pointing to. That description is a bit wordy, so here are some examples.

- `intArray: 0x0`
  `size: 4`
  `ptr: 0x0`
  `return: 1`
- `intArray: 0x0`
  `size: 4`
  `ptr: 0xF`
  `return: 1`
- `intArray: 0x0`
  `size: 4`
  `ptr: 0x10`
  `return: 0`
- `intArray: 0x100`
  `size: 30`
  `ptr: 0x12A`
  `return: 1`
- `intArray: 0x100`
  `size: 30`
  `ptr: 0x50`
  `return: 0`
- `intArray: 0x100`
  `size: 30`
  `ptr: 0x18C`
  `return: 0`

Please post on Piazza if you are having trouble understanding any of these examples!

Byte Traversal

The next two questions in `pointer.c` have you reading and writing data by understanding the layout of the bytes.

In C strings do not have knowledge of how long they are. In order to find out we must calculate it for ourselves. All strings in C are arrays of characters that end with a null terminator character - `'\0'` . The `stringLength` function has you return the length of a string, given a pointer to its beginning. You **are** allowed to use loops for this one. Also note that the null terminator character does ***NOT*** count as part of the string length.

The `endianExperiment` function has you set the value a pointer points to to the number 351351. Remember that we work with little endian data storage, and what that means.

Selection Sort

The final part of the lab has you implement `selectionSort`. Selection sort works by effectively partitioning an array into a sorted section, followed by an unsorted section. It repeatedly finds (and selects) the minimum element in the unsorted section, and moves it to the end of the sorted section (`swapInts` might be useful for this). The pseudo code might look something like this:

```
arr - an array
n - the length of arr

for i = 0 to n - 1
  minIndex = i
  for  j = i + 1 to n
      if arr[minIndex] > arr[j]
          minIndex = j
      End If
  Swap(arr[i], arr[minIndex])
  Next j
Next i
```

Note that you **are** allowed to use loops and if statements in this one.

Checking Your Work

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

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

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

  This only checks if your solutions return the expected result. *We may test your solution on inputs that ptest does not check by default and we will review your solutions to ensure they meet the restrictions of the assignment.*

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

```
$ python dlc.py
```

- The `dlc` program enforces a stricter form of C declarations than is the case for C++ or that is enforced by `gcc`. In particular, in a block (what you enclose in curly braces) all your variable declarations must appear before any statement that is not a declaration. For example, `dlc` will complain about the following code:

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

  Instead, you must declare all your variables first, like this:

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

- The `dlc` program will also complain about binary constants such as `0b10001000`, so avoid using them.

Overview
Setup
Instructions
Using Pointers
Checking Work
Reflection
Submission

Lab 1a Reflection

Make sure your answers to these questions are included in the file `lab1Areflect.txt`!

In both `lab0.c` and `pointer.c`, we saw the effects of pointers and pointer arithmetic:

1. Briefly describe why pointer arithmetic is *useful/beneficial* (not just its definition). [3 pt]
2. Think about how you calculated the actual difference (in bytes) between two addresses in C *without any compiler warnings*. Briefly explain why each step was necessary? [3 pt]
3. Notice that the parameters to the function `swapInts` were both pointers. Explain why this is necessary. What would happen if the parameters were integers? [3 pt]

Submission

Please submit your completed `pointer.c` and `lab1Areflect.txt` files to the assignments page (../submit.php).