CPSC 213 – Assignment 3

Static Variables and C Pointers

Due: Friday, January 27, 2016 at 11:59pm
After an 12-hour, no-penalty grace period, no late assignments accepted.

Overview

This assignment has three parts. First you will examine two C programs that access arrays and translate them into assembly language. Then you will answer some questions about C pointer arithmetic. Finally, you will investigate dynamic arrays and pointers in C by writing a bit of C code.

As with all assignments, you are encouraged to do this in groups of two. Just be sure that you both do all of the work, helping each other. Don't split up the work so that you both do half … unless, of course, you're shooting for a 50% on the midterm :).

Questions1 and 2: Writing Assembly *[48%]*

The code file for this week is found here: *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a3/code.zip*.

This file contains an directory named `examples` that contains a set of example C programs and their sm213 assemble-code implementation. The code file also contains two C files named `q1.c` and `q2.c`.

Start with the examples. Carefully examine each of them. Run the assembly versions in the simulator, step by step. Compare their execution to the C versions. Get comfortable with this translation from C to assembly and with how assembly executes in the machine.

Now, that you're getting comfortable, take the first of the C files that don't have assembly, `q1.c`, and produce your own sm213 file, `q1.s`, that does what this program does; just the parts of it that are indicated in the C file. Note that you can run the C program, specifying input values on the command line to see what the program should do. In the simulator you'll enter the inputs directly in the memory locations of variables and you'll examine the output the same way.

Every line of your assembly file must have a comment. The comment should explain what the line does by referencing C syntax whenever possible. For example, if an assembly instruction

loads the value of a variable into a register, the comment should name that variable. Do the same for the other C file.

Be sure to end your programs with a "`halt`" instruction so that the simulated CPU stops when it reaches the end of your program. If you don't do this, then it will keep running, interpreting whatever if finds in memory as instructions, probably doing very strange things.

We suggest that you start this by translating only the first line of C into assembly. Then, test this assembly in the simulator. Be sure it works, then add the second line and repeat. This is the best way to implement any program. You should try to almost always have something that runs and that you've tested. Make small changes and re-test. Don't try to write the whole then and then test it.

Now do the same for `q2.c`, translating it into `q2.s`.

How to Compile a C program

A C program consists of one or more files with extensions ".c" or ".h". Files ending in ".c" are C source code. Files ending in ".h" are header (i.e., interface) files. Every program has at least one ".c" file. Some will use multiple ".c" files and the ".h" files will describe the interface each of them provides to the other. To import an interface, a ".c" file will include initial lines of the form '`#include "foo.h"`'. In this assignment, every program consists of a single ".c" file and so there are no ".h" files. Subsequent assignments will be more complex.

There are many dialects of C out in the wild. In this course we will use the *2011 GNU* dialect, which is common (and the default on MacOS, but not on Linux). Dialects differ slightly in syntax and so you have to tell the compiler which dialect you are intending to use. You do this by adding the string "`-std=gnu11`" as a compiler option (note `11` is eleven, as in 2011).

Another thing you have to tell the compiler is the name of the program you want it to create. You do this with the "`-o foo`" option (to make a program called `foo`). If you don't include this option, the compiler places the output in a file called `a.out`.

So, lets say you want to compile `e1.c` to produce an executable named `e1`. You would type the following at the command line.

```
gcc -std=gnu11 -o e1 e1.c
```

Questions 3-5 *[16%]*

Answer These Questions

Place your answers in files named `q3.txt`., `q4.txt`, and `q5.txt`. Note that if you aren't sure of the answers, you can always write C program and have it tell you the answer.

Assume that the array `a` is declared and initialized like this:

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
```

3. What is the value of: `*(a+3)`?
4. What is the value of: `&a[7] - &a[2]`?
5. What is the value of: `*(a + (&a[7]-a+2))`?

Question 6 *[36%]*

Overview

In *http://www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a3/code.zip*, you will find a file named `bubble_sort_static.c` that takes as input up to 4 integers on the command line and uses the *Bubble Sort* algorithm to sort them.

You compile this program like this (again that's "gnu eleven")

```
gcc -std=gnu11 -o bubble_sort_static bubble_sort_static.c
```

You run it like this:

```
./bubble_sort_static 78 3 43 1
```

Read this C file carefully.

Just like `q1.c` and `q2.c`, this program's `main` function has two parameters: `argc` and `argv`. In Java it is similar: `main` has one parameter, an array of strings. That's what `argc` and `argv` are: `argc` is the number of stings found on the command line that ran the program (5 in the case of the example) and `argv` is a dynamic array that contains each of these strings. In C, a string is simply an array of `char` (i.e., a `char*`) that is terminated by the first `null` (i.e., `0`) in the array.

You will also see that `main` copies these values into a static global array named `val`. This array is statically assigned a length of 4 and so the input is limited to 4 numbers.

Finally, once `main` has copied the input values into `val`, it calls `sort` to sort the values and then prints out the result.

Step 1: Modify the C program to use a dynamic array [18%]

The first step is to create a new C program named `bubble_sort_dynamic.c` that removes the static-array limitation of the original version. The only change you need to make is to turn `val` into a dynamic array and to allow the program to sort arbitrary lists of integers provided on the command line (e.g., more than 4).

Recall that you need to allocate dynamic arrays dynamically by calling the procedure `malloc`. For example to allocate an array of 10 shorts you say:

```
short s* = malloc (10 * 2);
```

Be sure to test your program.

Step 2: C Pointers [18%]

Now, create another C program called `bubble_sort_awesome.c` that extends your earlier work to remove all square brackets from your program. All access to the arrays must be made using pointer arithmetic and the dereference operator (i.e., "`*`"). Test this too.

What to Hand In

Use the `handin` program; the command-line version is strongly preferred.

The assignment directory is `~/cs213/a3`, it should contain the following ***plain-text*** files.

1. `README.txt` that contains the name and student number of you and your partner
2. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). Your partner should not submit anything.
3. `q1.s` and `q2.s`, your code for Questions 1 and 2.
4. q3`.txt`., q4`.txt`., and q5`.txt` with your answers to Questions 3-5.
5. `bubble_sort_dynamic.c` and `bubble_sort_awesome.c`.