Lab Assignment 3: Static Variables and C Pointers

Due Sep 28, 2018 by 11:59pm **Points** 100 **Available** after Sep 22, 2018 at 12pm

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 :).

Questions 1 and 2: Writing Assembly [48%]

The code file for this week is found here: **a3code.zip**.

This file contains an directory named `examples` that contains a set of example C programs and their sm213 assemble-code implementation. It also contains a `Makefile` that will compile all of the example C programs in that directory. See the section below for more details.

Start by compiling the examples by navigating to that directory from the command prompt and typing the command `make`. This command will read the `Makefile` and use it to determine how the program should be compiled and what commands must run. You can then now run the C versions of `e1`, `e2`, `e3` and `e4`. Carefully examine each program to understand their behaviour and then run the assembly versions in the simulator, step by step. Get comfortable with how the `Makefile` works, the translation from C to assembly and how assembly executes in the machine.

Configuring a Makefile

In any language, large programs consists of a collection of program and library files that must be compiled and combined (i.e., *linked*) to form the program's executable file. And so it becomes important to be able to specify how these pieces fit together and how they depend on each other so that the executable can be built automatically when you change one of its components.

Integrated development environment tools like IntelliJ do this for you behind the scenes. Some systems use a tool called *Ant* to do this (Ant was developed to build the Apache web server and is now a stand-alone, open-source tool from Apache).

From its origins, Unix systems included a configuration management tool called `make` (current Linux systems use a version of this tool created by GNU called *gnu make*). To use make, you create a file called `Makefile` that describes a program's configuration and then, when you want to build it, you type `make` at the command-line instead of typing a long gcc command.

`Makefile` syntax is fairly simple, but not that intuitive. It consists mainly of statements of this form:

```
blah: part1 part2 part3
     command_to_build_blah_from_its_parts
```

This says that *blah* depends on three other files — *part1*, *part2*, and *part3* — and if any of them change, then you can rebuild blah from these parts using the command provided. Note that the command line must be preceded by a TAB character (not spaces) to work. The command line is optional if there is already a default rule defined for building that type of target (e.g., creating an executable based on a C file with the same name). You can define new default rules using the make wildcard character, %, in place of a name. You can also define variables. Some variables, such as CFLAGS are used by default rules and so you can change them to change the behaviour of these rules. You can define your own variables and use them using the `$(var)` syntax.

For example, if you have a C source file named `prog.c` and you wanted to compile it with a certain set of command line options to produce an executable named `prog`, you might place the following in the file called `Makefile` in the directory that contains your source code.

```
CFLAGS += -std=gnu11 -g -Wall
EXES = prog
OBJS = prog.o

all: $(EXES)
clean:
    rm -f $(OBJS) $(EXES)
prog: prog.o
prog.o: prog.c
```

Having done this you can build the program by typing either "`make`", "`make all`" or "`make prog`" in a terminal inside the directory containing the Makefile. Or you can delete the executable and intermediate files by typing “`make clean`”. Note that each time you ask make to make a specific target, make searchers for that target on the left-hand-side of colon and then builds it recursively. When a target file already exists, make compares its last-modification-time to that of the files it depends on, and only rebuilds the target if they have been modified after the last time the target was changed.

We will use makefiles throughout the rest of the term, starting this week.

Translate q1.c and q2.c to Assembly

Now, that you are getting comfortable, create your own `Makefile` to compile the C programs in `q1.c` and `q2.c`. Compile the programs by typing `make`. Start by understanding the behaviour of `q1.c` and then produce your own sm213 file, `q1asm.s`, that has the same behaviour. Note that when you run the C program, you specify the input values on the command line. In the simulator you will enter the inputs directly in the memory locations of variables as is done in the examples provided (e1, e2, e3, e4).

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 have something that runs and that you've tested. Make small changes and re-test. Don't try to write the whole program and then test it.

Now do the same for `q2.c`, translating it into `q2asm.s`. In this file, in order to work with the autotest, you will need to assign a label to each allocation including one for `s_ar` even though your program instructions should not make use the `s_ar` label.

```
.pos 0x2000
a:   .long 0
     .long 0
     .long 0
s:   .long 0x3000
tos: .long 0
tmp: .long 0

.pos 0x3000
s_ar: …
```

For all of this work you can either use your implementation of the simulator from Assignment 2 or the reference implementation.

Translate update_dynamic.c to Assembly

Recall `update.c` from Assignment 2 that you translated to `update.s`. Both of these files have been provided in this assignment for your convenience. You have also been provided with `update_dynamic.c` which is the equivalent of update.c but with a dynamically allocated array.

You should translate `update_dynamic.c` to `update_dynamic.s`. As in Assignment 2, you are only to translate the lines of code within the update function.

As in `q2asm.s`, in order to work with the autotest, you will need to assign a label to each variable allocation including one for `array_data` even though your program instructions should not make use the `array_data` label:

```
.pos 0x1000
t:     .long 0x0
array: .long 0x2000

.pos 0x2000
array_data: …
```

Questions 3-5 [16%]

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%]

In the provided code, 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. Read this C file carefully.

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

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

Feel free to add this compilation rule to the Makefile you created in the first part of this assignment, if you don't want to compile with the long gcc command.

You run the program like this:

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

Just like `q1.c` and `q2.c`, this program's main function has two parameters: `argc` and `argv`. The `argc` value is the number of strings found on the command line that ran the program (5 in the case of the example) and `argv` is an array that contains each of these strings. In C, each 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 the value of the command-line arguments 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 (each 2-bytes long) you say:

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

Be sure to test your program. You must also edit your Makefile so that it compiles `bubble_sort_dynamic.c` into an executable called `bubble_sort_dynamic`.

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. You must also edit your Makefile so that it compiles `bubble_sort_awesome.c` into an executable called `bubble_sort_awesome`.

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.

- `README.txt`: contains the name and student number of you and your partner.
- `PARTNER.txt`: contains 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.
- `q1.c`, `q2.c`, `q1asm.s` and `q2asm.s`: your code for Questions 1 and 2.
- `update_dynamic` `.s`: your translation of `update_dynamic.c`.
- `q3.txt`, `q4.txt`, and `q5.txt`: your answers to Questions 3-5.
- `bubble_sort_dynamic.c` and `bubble_sort_awesome.c`.
- Your `Makefile`, containing rules for generating (at least) executables `q1`, `q2`, `bubble_sort_dynamic` and `bubble_sort_awesome`.