CPSC 213: Assignment 3

Last Modified: September 22, 2014 at 6:30pm

Due: Sunday, September 28, 2014 at 11:59pm

After an eight-hour grace period, no late assignments accepted.

Goal

The goal of this assignment is the learn more about structs in C and how they are implemented by the compiler. To begin you will convert a small Java program to C using structs. Then, you'll switch to the considering the translation from C to machine-code, in two steps. There is a new snippet to get you started. Then there is a small C program to convert to assembly.

Structs in C

Download the file www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a3/code.zip. It contains the following files that you will use for Part I of the assignment plus another file that will be described in Part II.

For Part I the files are:

- BinaryTree.{java,c}
- S4-instance-var.{c,java,s}

Converting Java to C

The file BinaryTree.java contains a Java program that implements a simple, ordered, binary tree. Examine this code. Compile and run it from the Unix command line (or in your IDE such as Eclipse) :

```
javac BinaryTree.java
java BinaryTree
```

The file BinaryTree.c is a skeleton of a C program that is meant to do the same thing. Using the Java program as your guide, implement the C program. The translation is pretty much line for line, translating Java's classes/objects to C's structs.

Note that since C is not object oriented, C procedures are not invoked on an object (or a struct). And so, you will see that Java instance methods when converted to C have an extra argument: a pointer to the object on which the method is invoked in the Java version (i.e., what would be "**`this`**" in Java).

Of course, C also doesn't have "**new**", for this you must use "**malloc**". It doesn't have "**null**", for this you can use "**NULL**" or "**0**". Finally, C doesn't have "**System.out.printf**", use "**printf**" instead.

Compile and test your implementation of BinaryTree.c on the command line:

```
gcc -o BinaryTree BinaryTree.c
./BinaryTree
```

It is sufficient to show that your C program produces the same output as the Java program.

Snippet S4-instance-var

There is one snippet for this week. As you did last week, load this snippet into the SM213 and single step through its execution. Turn on animation and run it slowly. Slow the animation by hitting the "slower" button. Hit the "pause" button when you want to take a longer look. Carefully **summarize** what you observe.

- S4-instance-var

Convert C to Assembly Code

Now, combine your understanding of snippets S1, S2 and S4 to answer the following questions about this piece of C code.

```
struct S {
  int       x[2];
  int*      y;
  struct S* z;
};

int      i;
int      v;
struct S s;

void foo () {
  v = s.x[i];
  v = s.y[i];
  v = s.z->x[i];
}
```

1. Implement this code in SM213 assembly, by following these steps:

   a. Create a new SM213 assembly code file called **a3.s** with three sections, each with its own **.pos**: one for code, one for the static data, and one for the "heap". Something like this:

```
 .pos 0x1000
code:

.pos 0x2000
static:
```

```
.pos 0x3000
heap:
```

b. Using labels and .long directives allocate the variables **i**, **v**, and **s** in the static data section. Something like this (the ellipsis indicates more lines like the previous one) :

```
 .pos 0x2000
static:
i:    .long 0
v:    .long 0
s:    .long 0
      .long 0
      ...
```

c. Implement the three statements of the procedure **foo** (not any other part of the procedure) in SM213 assembly in the code section of your file. Comment every line carefully.

d. Initialize the variable **s.y** to store a pointer to the beginning of the "heap"section, as if the program had called "**malloc**" to allocate an array of 3 integers. What you are doing here is modelling some of the dynamic calculation of the program (the malloc and initialization of **s.y**) so that you can test the code you have written. Something like this:

```
 .pos 0x3000
heap0: .long 0
       .long 0
       .long 0
```

e. Initialize the variable **s.z** to store a pointer to the next available part of the "heap" section (i.e., right after the three ints of heap0). This part of the heap should have one .long for every element of struct S. Something like this:

```
heap1: .long 0
       .long 0
       ...
```

f. Test your code for a few different values of **i**, **s.x[0..1]**, **s.y[0..2]**, and **s.z->x[0..1]**.

2. Use the simulator to help you answer these questions about this code. The questions ask you to count the number of memory reads required for each line of **foo()**. When counting these memory reads do not include read for variable **i**.
   a. How many memory reads occur when the first line of **foo()** executes? Explain.
   b. How many memory reads occur when the second line of **foo()** executes? Explain.
   c. How many memory reads occur when the third line of **foo()** executes? Explain.

Material Provided

In the file code.zip:

1. BinaryTree.{java,c}
2. S4-instance-var.{java,c.s}

What to Hand In

Use the **`handin`** program. The assignment directory is **a3**, it should contain the following files (and nothing else).

1. A single file called “README.txt” that includes your name, student number, four- or five-digit cs-department undergraduate id (e.g., the one that’s something like a0b1), and all written material required by the assignment as listed below.
2. BinaryTree.c
3. Your observations from running snippet S4.
4. a3.s
5. Answers to the questions 2a, 2b, and 2c in README.txt