Lab Assignment 4: Structs and Instance Variables

Due Oct 5, 2018 by 11:59pm **Points** 100 **Available** after Sep 29, 2018 at 12pm

Overview

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 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.

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. Concepts in this assignment will be valuable knowledge in the midterm.

Background

Some notes about C programs that you may find helpful. Some of this is repeated from Assignment 3, included again here for context.

Parts of a C program

As you saw last week, C programs typically consist of a collection of ".c" and ".h" files. C source code files end in ".c". Files ending in ".h" are called header files and they essentially list the public interface to a particular C file. In this assignment you will mostly ignore header files. You will create only a ".c" source file. However, in order to call library functions such as `malloc()` you need to include some standard system header files in your program.

To include a header file in a C program you use the `#include` directive. This is much like the import statement in Java. What follows the directive is the name of a header file. Header files delimited by `<>` brackets are standard system files; those in quotes are your own header files that are typically co-located with your .c code. For this assignment you need only include two standard header files, so you'll need the following lines at the beginning of your file (this is already done for you).

```
#include <stdlib.h>
#include <stdio.h>
```

The first line gives you access to `malloc` (among other functions) and the second to `printf` (among other functions).

Debugging C

You may need to debug your C program. To use the debugger, you need to add the `-g` flag when you compile your program. Make sure this flag is part of the `CFLAGS` variable on your `Makefile`.

On Linux and Windows, you debug with a program called "`gdb`"; on Mac a similar program is called "`lldb`". In either case, to start your program in the debugger, you type the name of the debugger, a space, and then the name of your executable, like this

```
gdb ./foo # or lldb ./foo
```

Now you might want set a breakpoint type `b` or `break` and a line number or procedure name before you run the program with `r` or `run`. You can examine the value of variables with `p` or `print` followed by the variable or expression. You can step through the execution of a procedure with `n` or `next`, which skips over procedure calls, or if you want to step into a procedure call use `s` or `step`. To continue running until the next breakpoint (or until the end of the program) type `c` or `continue`. To learn more, type `h` or `help`.

What You Need to Do

Question 1: Debugging a Simple C Program [10%]

The first thing to do is create a very simple C program, compile it and run it. Using the editor of your choice, create a file called `simple.c` that looks like this:

```
#include <stdlib.h>
#include <stdio.h>

void foo (char* s) {
    printf ("Hello %s\n", s);
}

int main (int argc, char** argv) {
    foo ("World");
}
```

Then compile it using the gcc command or make (instructions in previous assignments), and type the following command to run it:

```
./simple
```

Then run the debugger, set a breakpoint in `foo`, run it until it hits the breakpoint, print the value of `s`, and then continue its execution to completion. When you print `s`, the debugger understands that this may be a pointer to a null-terminated string and so it prints the value of that string. You would probably run commands like these (note that `(gdb)` is the gdb prompt, and you should not type it):

```
gdb simple
(gdb) b foo
(gdb) r
(gdb) p s
(gdb) c
(gdb) quit
```

Record what the program prints in file `q1d.txt` and the value of `s` you see at the breakpoint in the file `q1s.txt`.

Question 2: Convert Java Program to C [50%]

Download the file **a4code.zip**. It contains two files you need for Question 2 plus additional files that you will use later. The files you need for this part are `BinaryTree.java` and `BinaryTree.c`.

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

```
javac BinaryTree.java
java BinaryTree 4 3 2 1
```

When the program runs, the command-line arguments (in this case the number 4 3 2 1) are added to the tree and then printed in depth-first order based on their value. You can provide an arbitrary number of values on the command line with any numeric values you like.

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 and objects to C's structs.

Note that since C is not object-oriented, C procedures are not invoked on an object (or a struct). Instead, 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`. Note that `malloc` only allocates memory; it does not do the other things a Java constructor does such as initialize instance variables. C also doesn’t have `null`; for this you can use `NULL` or 0. Finally, C doesn’t have `System.out.printf`, you can use `printf` instead.

Your goal is to have the C program produce the same exact output as the Java program for any inputs. You must also create a makefile to compile your program with `CFLAGS = -std=gnu11 -g`.

You Might Follow these Steps

1. Start by defining the Node struct. Note that like a Java class, the struct lists the instance variables stored in a node object; i.e., value, left, and right. Note that in Java left and right are variables that store a reference to a node object. Consult your notes to see how you declare a variable in C that stores a reference to a struct.
2. Now write the create procedure that calls `malloc` to create a new struct Node, initializes the values of value, left, and right, and returns a pointer to this new node. Then, in the main function, call this procedure to allocate one node with the value 100 and declare a variable `root` to point to it.
3. At this point you have the code that creates a tree with one node. Now write the procedure `printInOrder` and compile and test your program to print this one node. Do not proceed to the next step until it works.
4. Now implement `insert`. And test it by inserting, in the main function, two nodes to `root`: one with value 50 and one with value 150. So, now you have a tree with three nodes. When you call `printInOrder` on `root` you should get the output: `50 100 150`.
5. At this point you should be ready to complete the implementation of main to insert nodes into the tree with values the come from the command line instead of these original, hardcoded values. Test it again and celebrate.

Snippet S4-instance-var

The file you downloaded in Question 2 also contains the files `S4-instance-var.java`, `S4-instance-var.c` and `s4-instance-var.s`. Carefully examine these three files and run `s4-instance-var.s` in the simulator. Turn animation on and run it slowly; there are buttons that allow you to pause the animation or to slow it down or speed it up. Trace through each instruction and try to get comfortable with what each is doing and how the instructions relate to the .c code. Once you have a good understanding of the snippet, you can move on to Question 3. There is nothing to deliver in this step.

Question 3: Convert C to Assembly Code [40%]

Now, combine your understanding of snippets S1 and S2 (from previous assignments) and S4 to do the following with this piece of C code.

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

int i;
int v0, v1, v2, v3;
struct S s;

void foo () {
  v0 = s.x[i];
  v1 = s.y[i];
  v2 = s.z->x[i];
  v3 = s.z->z->y[i];
}
```

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

1. Create a new SM213 assembly code file called `q3.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:
   ```

2. Using labels and `.long` directives allocate all global variables in the static data section. Note the variable s is a `struct S` and so to allocate space for it here you need to understand how big it is. This section of your file should look something like this (the ellipsis indicates more lines like the previous one):

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

3. Now initialize the locations `s.y`, `s.z`, `s.z->z`, and `s.z->z->y` to point to locations in heap as if malloc had been called for each of them. For the array, allocate two integers. You want to create a snapshot of what memory would look like after the program executed these statements:

```
s.y       = malloc (2 * sizeof (int));
s.z       = malloc (sizeof (struct S));
s.z->z    = malloc (sizeof (struct S));
s.z->z->y = malloc (2 * sizeof (int));
```

   You will need to assign labels to each of these things, like this:

```
s: .long 0
   .long 0
   .long s_y
   .long s_z
...
heap:
s_y: .long 0 # s.y[0]
     .long 0 # s.y[1]
s_z: .long 0 # s.z->x[0]
     .long 0 # s.z->x[1]
     .long 0 # s.z->y
     .long s_z_z # s.z->z
s_z_z: …
```

4. Implement the four 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. NOTE: you cannot use any dynamically computed values as constants in your code. So, for example, you can not use the labels `s_y`, `s_z` etc. but they should be included in your `.s` to make it easier for the auto test to give you feedback.
5. Test your code.

Once you have implemented the function, 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 the read for variable `i`. Write the answers in files `q3a.txt`, `q3b.txt`, `q3c.txt` and `q3d.txt`, respectively.

a. How many memory reads occur when the first line of `foo()` executes?
b. How many memory reads occur when the second line of `foo()` executes?
c. How many memory reads occur when the third line of `foo()` executes?
d. How many memory reads occur when the fourth line of `foo()` executes?

What to Hand In

As usual, use the handin program to submit your answers. The assignment directory is `~/cs213/a4`, it should contain the following files (and nothing else):

1. `README.txt`: 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. For Question 1: `q1` `d.txt` and `q1s.txt`.
4. For Question 2: `BinaryTree.c` and `Makefile`.
5. For Question 3: `q3.s`, `q3a.txt`, `q3b.txt`, `q3c.txt`, and `q3d.txt`. The text files should contain your answers as numbers without any explanation to make it easy for the auto-marker to read your

answer. Additionally, `q3.s` should include the labels: `s_y`, `s_z`, etc. with your variable allocations.