CPSC 213 – Assignment 4

Structs and Instance Variables

Due: Wednesday, February 3, 2016 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.

Background

Some notes about C programs that you may find helpful…

Parts of a C program

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 other header files that are typically co-located with your `.c` code. For this assignment you need only include two standard header files, by putting the following as the beginning of your file (this is already done for you).

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

This will give you access to `malloc` and `printf`.

One other thing. As in Java, the procedure called `main` is special. This is the first procedure that runs when you execute your a program.

Creating and Editing a C Program

You need to decide where you will write, compile and test your programs and what editor and/or IDE you will use. Any ***plain-text*** editor will work fine (e.g., emacs, vim, TextEdit or NotePad). If you use an editor designed to produce formatted text, be sure your file is configured to be in plain-text mode; be careful, this is often not the default setting. The compiler does not understand rich-text format. If you attempt to compile a file and get errors complaining about unknown characters, then you've probably saved your file in non-plain text.

It is easy for you to see how the compiler sees your program to test that you have it in plain text. At the UNIX command line type

```
cat foo.c
```

To see the content of the file `foo.c`. If you see strange characters and so will the compiler.

Compiling C

To compile a C program you typically need access to the UNIX command line (or Cygwin). The command is called `gcc`. Be sure to use `gcc` and not `g++`, which is the C++ compiler. Enter `gcc` and then follow that with a specification of the language variant you are using. Examples I've given in class use the *gnu eleven* (i.e., 2011) standard, which you can specify with "`-std=gnu11`" **(i.e., gnu eleven, not gnu ell ell).** Then you should include "`-o foo`" to specify the name of the output file (in this case "`foo`"). If you don't include this option, the compiler will create a file called "`a.out`". Then after this option you list the name of the C file to compile. So, for example, if you want to compile the file `foo.c` into the executable `foo`, type:

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

To run a program you complied you type that name on the command line preceded by `./`. So, for example, if you want to run `foo` you type:

```
./foo
```

Debugging C

You *may* need to debug your C program. To use the debugger, you need to add "`-g`" when you compile your program. Like this

```
gcc -g -o foo foo.c
```

On Linux and Windows, you debug with a program called "`gdb`"; on the Mac its 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 like this

```
lldb foo
```

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

What You Need to Do

Part 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 ("%s World\n", s);
}

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

Then type the following command to compile it (note that the `11` in `gnu11` below is the number eleven):

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

Then type the following command to run it:

```
./simple
```

Then type the following command to re-compile it for debugging (you can just include -g all the time if you like):

```
gcc -std=gnu11 -g -o simple simple.c
```

Then type the following (you type what is in black) to 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:

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

If you are on a Mac then use the command `lldb` instead of `gdb`.

Record what the program prints file `P1d.txt` and the value of `s` you see at the breakpoint in the file `P1s.txt`.

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

Download the file www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a4/code.zip. It contains two files you need for Part 2 plus additional files that you will use later. The files you need for this part are:

- `BinaryTree.java`
- `BinaryTree.c`

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

```
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`". Note that all that `malloc` does is allocate 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`", use "`printf`" instead.

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

```
gcc -std=gnu11 -o BinaryTree BinaryTree.c
./BinaryTree
```

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

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 you might want to write a *constructor* procedure that calls `malloc` to create a new `struct Node`, initializes the values of `n->value`, `n->left`, and `n->right`, and

returns a pointer to this new node. Then call this procedure to allocate the node with value `100` and assign this to the variable `root`, which like `left` and `right` is a pointer to a struct. You might name this procedure something like `newNode()`; note that C does not have native constructors and so what you are doing here is simply writing a procedure that acts like the constructor. Or you could just do all of these steps inline as the first few statements of `main`.

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. Do not proceed to the next step until it works.
4. Now implement `insert` and `insertNode`. Note that these procedures work together. The `main` calls `insert` to insert a value. And then `insert` creates a node to store the value and calls `insertNode` to insert that node into the tree. If you wrote a constructor procedure, call it from `insert`. If not, then repeat the inline steps you used in `main` to create the first node.
5. Modify `main` to insert one node and test your code. Once that works add the rest of the `insert` calls.

Part 3: Snippet S4-instance-var

The `code.zip` file you downloaded in Part 2 also contains the files

- `S4-instance-var.java`
- `S4-instance-var.c`
- `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 explain to yourself what each is doing and how the instructions related to the `.c` code. Once you have a good understanding of the snippet, you can move on to Part 4. There is nothing to hand in for Part 3.

Part 4: Convert C to Assembly Code [40%]

Now, combine your understanding of snippets S1, S2 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      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 `a4.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
   heap0:
   ```

   (b) Using labels and `.long` directives allocate the variables `i`, `v`, and `s` 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 its. 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
   v:     .long 0
   s:     .long 0
          .long 0
          ...
   ```

   (c) 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 2 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
   ```

   (d) Initialize the variable `s.z` to store a pointer to the next available part of the "heap" section (i.e., right after the two ints of `heap0`). This part of the heap should have one `.long` for every element of struct S. Do no further dynamic allocation; set the values of `s.z->y` and `s.z->z` (the dynamic array and struct pointers) to zero. Your code should like something like this (again … means more `.long`'s as needed):

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

(e) 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.

(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.

What to Hand In

Use the `handin` program. The assignment directory is `~/cs213/`**`a4`**, it should contain the following files (and nothing else).

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. `P1d.txt` that contains your Part-1 description of what the program prints.
4. `P1s.txt` that contains the value of `s` at the breakpoint and nothing else.
5. `BinaryTree.c`
6. `a4.s`
7. `Q2a.txt`, `Q2b.txt`, and `Q2c.txt` containing your answers to the questions in Part 4 without any explanation — make it easy for the auto-marker to read your answer.
8. `Q2a-desc.txt`, `Q2b-desc.txt`, and `Q2c-desc.txt` containing your explanations for these three questions.