CPSC 213 – Assignment 4

Structs and Instance Variables

Due: Saturday, October 5, 2019 at 11:59pm
After an 8-hour, no-penalty grace period, no late assignments accepted.

Learning Objectives

Here are the learning objectives of this assignment, which will be examined in this week's quiz. They are a subset of the unit learning objectives listed on Slide 2 of Unit 1c.

After completing this assignment you should be able to:

1. read and write C code that includes structs
2. compare Java classes/objects with C structs
3. explain the difference between static and non-static variables in Java and C
4. distinguish static and dynamic computation for access to members of a static struct variable in C
5. distinguish static and dynamic computation for access to members of a non-static struct variable in C
6. translate C struct-access code into assembly language
7. count memory references required to access struct elements

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. Some of this is repeated from Assignment 3, included again here for convenience.

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 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. 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`".

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* (updated to *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 `gcc`.

`Makefile` syntax is fairly simple, but strange. It consists mainly of statements of this form:

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

Note: This second line (the rule) must have a leading TAB, not spaces.

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. The second line is optional if there is already a default rule defined for building that type of target (defined by its file extension; e.g., `.c`). 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
EXES    = prog
OBJS    = prog.o

all:  $(EXES)
clean:
     rm -f $(OBJS) $(EXES)
tidy:
     rm -f $(OBJS)

prog: prog.o
prog.o: prog.c
```

Having done this you can build the program by typing either "`make`" or "`make prog`". You can delete all intermediate files by typing "`make tidy`". And, you can delete everything but the original source code 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 its dependents to see if target needs to be rebuilt, and only rebuilds it if necessary.

We will use makefiles throughout the rest of the term, starting this week with Question 2.

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 ("%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 the variable `s` at this point in the execution, 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 it thus prints the value of that string. What we want is the value the string.

```
gdb simple
(gdb) b foo
(gdb) run
(gdb) print s
(gdb) cont
```

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

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 *www.students.cs.ubc.ca/~cs-213/cur/assignments/a4/code.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`
- `BinaryTree.c`

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

```
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/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 "`out.printf`", use "`printf`" instead. Your goal is to have the C program produce the same exact output as the Java program for any inputs.

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 call this procedure to allocate one node 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 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, hard-coded values 50, 100, and 150. Test it again and celebrate.

Snippet S4-instance-var

The `code.zip` file you downloaded in Question 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 Question 3. There is nothing to hand in this step.

Question 3: 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      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];
}
```

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

   (a) 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:
   ```

   (b) Using labels and `.long` directives allocate the variables `i`, `v0`, `v1`, `v2`, `v3`, 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
v0:   .long 0
v1:   .long 0
v2:   .long 0
v3:   .long 0
s:    .long 0
      …
```

(c) Now initialize the variables `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 and 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 probably want to assign labels to each of these things, something like this:

```
s:     .long 0      # s.x[0]
       .long 0      # s.x[1]
       .long s_y    # s.y
       .long s_z    # 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: …
```

Where everything after the `heap` label are things that would have been allocated by `malloc`.

(d) 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 can not use any dynamically computed values as constants in your code. So, for example, you can not use the labels `s_y`, `s_z` etc.

(e) Test your code.

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 the read for variable `i`.

(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

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

1. `PARTNER.txt` containing your partner's CWL login id and nothing else. Your partner should not submit anything.
2. For Question 1: `q1d.txt` and `q1s.txt`. Be sure that `q1s.txt` contains the value of `s` at the breakpoint and nothing else.
3. For Question 2: `BinaryTree.c` and `Makefile`
4. For Question 3: `q3.s`, `q3a.txt`, `q3b.txt`, `q3c.txt`, and `q3d.txt` containing your answers to the questions in Part 2 of that question. The text files should contain your answers as numbers without any explanation to make it easy for the auto-marker to read your answer.