CPSC 213 – Assignment 4

Structs and Instance Variables

Due: *Due to the Thanksgiving holiday:* Tuesday, October 13, 2020 at 11:59pm
After an 8-hour, no-penalty grace period, no late assignments will be 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 to 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 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 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 then 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. We typically use the *gnu eleven* (i.e., 2011) standard in class, which you can specify with "`-std=gnu11`". 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 the compiled program, you type the path to that program; for a program in the current directory, use `./`. So, for example, if you want to run `foo` you type:

```
./foo
```

Note that if you do *not* include a path (here, `./`) then your shell will look for a built-in system binary called `foo`, which will probably not exist.

Debugging C

You *may* need to debug your C program. In order to effectively use the debugger, you should tell the compiler to insert debugging information into the program by adding "`-g`" when you compile your program. Like this:

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

On Linux and Windows, you debug with a program called "`gdb`"; on the Mac it's 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
```

In the debugger, you can use various commands to control your program's execution and examine its state. A summary of commands is as follows (and you can use `help` for more info):

- `b <location>`: Set a *breakpoint* in a particular function or line number. The debugger will automatically pause the program when it reaches a breakpoint. For example, `b main` to set a breakpoint on main, or `b test.c:13` to set a breakpoint at line 13 of test.c.
- `run [arguments]`: Run the program, optionally providing arguments (as if it was run on the command-line). The program will start running immediately, and will only stop at a breakpoint.
- `bt`: Print out a back-trace, which is similar to a stack trace in Java, showing the function calls that lead to the current point in the program. Often, when debugging crashes, the crash will be in library code but the error is in user program code higher up in the back-trace.
- `p <variable>`: Print out the current value of the given variable or expression. For example, `p str` to print out the value of variable "`str`", or `p s1->d` to print out the `d` member of the dynamic structure `s1`.
- `s`, `n`: these commands step the program one line at a time. `s` (`step`) steps *into* function calls (*e.g.* stepping on the line `printf("foo")` will step into each line of printf), while `n` (`next`) steps to the *next* line of code, stepping *over* any function calls (letting the function call run to completion before pausing on the next line of the calling function).
- `c/continue`: Continue running the program. The program will stop at the next breakpoint or otherwise run until it crashes or exits.

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.

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`.

Using *make* has two principle benefits. First, by creating a `Makefile`, you specify the exact steps needed to build a program, which helps with reproducibility. Second, *make* will intelligently rebuild only the parts of a program which have changed (based on the source files which have changed), which can save a lot of time when modifying part of a larger program.

make has some *default* rules for compiling C programs. Thus, often times you can type simply `make foo` on the command-line to build `foo` from `foo.c`. But, you'll often want to write a `Makefile` in order to customize how it builds programs.

`Makefile` syntax is fairly simple:

```
target: input1 input2 input3
     command_to_build_output_from_its_inputs
```

Note: The second line (the command) must have a leading TAB.

This says that `target` depends on three other files — `input1`, `input2`, and `input3` — and if any of them change, then you can rebuild `target` from these parts using the command(s) provided. The command 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 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, to compile the program in Q1 of this assignment (`greet`) from the source file `greet.c`, you might place the following in the file called `Makefile` in the directory that contains the source code (this file is included in the code handout):

```
CFLAGS += -std=gnu11 -g
EXES    = greet

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

greet: greet.c
# don't treat all and clean as file targets
.PHONY: all clean
```

NOTE!! One of the quirks of *make*, being an old UNIX program, is that it *only allows **tabs** for indentation*, not spaces! Unfortunately, many editors automatically replace tabs with spaces,

which will break Makefiles. In your editor configuration, be sure to turn *off* any options like "replace spaces with tabs", "auto-expand tabs", and so forth for Makefile. If you've accidentally used spaces for indentation, you'll get a weird error like this when you go to run `make`:

```
Makefile:7: *** missing separator.  Stop.
```

Note also the line `.PHONY: all clean`; this line tells *make* that `all` and `clean` are not file targets. Without this line, if your directory happened to contain a file called `all`, the command `make all` would just do nothing.

Having written your `Makefile`, you can build the program by typing either "`make`" or "`make prog`". You can also delete everything but the original source code by typing "`make clean`". Without an argument, *make* builds the very first target in the file, which in our case is `all`. When you ask *make* to build a target, *make* searches for that target on the left-hand-side of a colon, enumerates its dependencies recursively, and then compiles any targets that have changed dependencies.

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

What You Need to Do

Download the file *www.students.cs.ubc.ca/~cs-213/cur/assignments/a4/code.zip.* It contains several files that you'll need for this assignment.

Question 1: Debugging a Simple C Program [30%]

The files you'll need for this part are `Makefile` and `greet.c`.

What we're going to do is compile a very simple C program, run it, and fix a bug in the program. Take a look at the file `greet.c` in the provided code package, reproduced below:

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

void greet(char *name, int id) {
  printf("Hello %s!\n", name);
  printf("  You are visitor #%s.\n", id);
}

int main() {
  greet("Kelly", 2000);
  greet("Morgan", 2001);
  return 0;
}
```

(a) Looking at the program, what is the *intended* output of the program? Submit this in the text file q1a.txt.

The accompanying `Makefile` will build the program `greet`. Thus, you can compile the program by running `make`. The *make* program will print out each command that it uses to compile the program. Note that if you run *make* twice in a row, the second time it won't do anything, because it knows your source file hasn't changed.

Run the program using the following command:

```
./greet
```

(b) After running the program, what did it actually print out? Submit this as the text file q1b.txt.

The last line of output that you see indicates that the program *crashed*. The operating system detected an invalid access to memory and killed the program. Your job is now to fix the bug that caused the crash. Load the program into a debugger using the following command:

```
gdb greet
```

(or `lldb greet` on macOS). Type **`run`** to run the program. `gdb` will tell you that the program "received signal SIGSEGV", indicating that the operating system flagged a memory error. If you're using lldb, you'll see a similar "stop reason = EXC_BAD_ACCESS" message.

Type **`bt`** to show a *back-trace*, showing you each function that was called up to the current point in reverse order - the most recent function calls are at the top. The first few calls will be from the C standard library "libc".

(c) In the back-trace, which line corresponds to the most recent function call from our program (i.e. *not* the standard library)? Submit the entire line in the text file q1c.txt.

Each line of the back-trace shows the function name, file name and (if known) the line number. Note that you might only see line numbers if the program was compiled with debugging information (`-g`).

Type **`b greet`** to set a *breakpoint* on the crashing function from our program.

Type **`run`** to rerun the program from the beginning (you may need to type `y` to confirm rerunning the program). Your program will pause right at the start of the `greet` function, and will show you the line of code that is about to be executed.

Type **`n`** to step to the next line of code, and keep entering **`n`** until your program crashes again.

(d) Which line of C code did our program run that caused the crash? Submit this line in the text file q1d.txt.

By finding out which line of code crashes, you have narrowed down the cause of the problem. This is an important step in debugging. Look at this line of code carefully.

(e) Formulate a hypothesis (a thoughtful guess) as to why this line of code crashes. Explain your hypothesis and reasoning in the text file q1e.txt.

With an idea of why it's crashing, you should be able to propose a fix to the program. Change the crashing line of code to make the program work as expected. *Hint:* a one character change is enough.

(f) Produce a fixed version of the code which compiles, runs, and successfully outputs what you expected it to output in part (a). Submit this line in the file greet.c.

Congratulations. If you got all the way through this, you've successfully debugged and repaired a program using a tried-and-true methodology: figure out the *expected behaviour*, acquire the *actual (incorrect) behaviour*, determine the *root cause*, and propose a *patch*. The same basic cycle is used to debug programs of any size, from tiny 10 line programs to 1,000,000 line megaprojects.

Question 2: Convert Java Program to C [30%]

The files you need for this part are `Makefile`, `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 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). Thus, Java instance methods 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.

Modify the provided Makefile to compile your new C program into the program `BinaryTree`.

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 that 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 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 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 hand in this step.

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

Now, combine your understanding of snippets S1, S2 (from the assignment 2 handout) 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". For example:

   ```
   .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 each `y` array, allocate two integers. You want to create a snapshot of what memory would look like after the program has 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 must assign labels to each of these things. In order for the auto-grader to recognize your solution you must follow our labeling scheme (i.e., use variable name for labels `i`, `v0`, `v2`, `v3`, and `s` and use labels like `s_z_z` to refer to `s.z->z`). For example, after the first `malloc` runs, maybe the memory looks like this:

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

(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 addresses as constants in your code. So, for example, you cannot use the labels `s_y`, `s_z` etc. in your code.

(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` (nor the instruction itself).

(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 `~/cs-213/a4`, and it should contain the following files (and nothing else).

1. For Question 1: `q1a.txt`, `q1b.txt`, `q1c.txt`, `q1d.txt`, `q1e.txt` and `greet.c`, containing your answers to parts (a) through (f) respectively.
2. For Question 2: `BinaryTree.c` and `Makefile`.
3. For Question 3: `q3.s`, containing your implementation of Part 1, and `q3a.txt`, `q3b.txt`, `q3c.txt`, and `q3d.txt` containing your answers to the questions in Part 2. The text files should contain your answers as numbers without any explanation to make it easy for the auto-marker to read your answer.