CS 252: Computer Organization

Assembly Project #4

Structs

due at 3pm, Sat 17 Apr 2021

1 Purpose

In this project, you will write several functions which use structs. You will read and write the fields of structs; you will also perform some searches through arrays of objects.

2 How Do Structs Work?

In this Project, we'll be working with structs. In C, a struct is simply a plan for how variables will be laid out inside an object. For instance, consider this struct, which is the one we'll be using in this project:

```
struct Turtle {
    char  x;
    char  y;
    char  dir;
    char *name;
    int   odometer;
};
```

This struct represents a very simplified "turtle" (`https://en.wikipedia.org/wiki/Turtle_graphics`). We won't be actually drawing anything on the screen - but we'll keep track of the turtle, and **pretend.**

The struct keeps track of the current position of the turtle - the `x,y` position, and the direction it's facing (0-3), representing North, East, South, West. It also has a name for each of the turtles, which of course is a pointer to a string.

Just to make things more interesting (and so the size wasn't a power of 2!), I added one more field: an "odometer." Every time that the Turtle moves (forward or backwards), the odometer increases by the distance that it travelled.

The fields of the struct are arranged as follows:

```
          +-------+
byte 0 |   x   |    <----- char x;
          +-------+
       1 |   y   |    <----- char y;
          +-------+
       2 |  dir  |    <----- char dir;
          +-------+
       3 | (pad) |    ... 'pad' bytes are not used for any field ...
          +-------+
       4 |       |
       5 | name  |    <----- pointer to the name of the turtle
       6 |       |
       7 |       |
          +-------+
       8 |       |
       9 | odom  |    <----- total of how far this Turtle has travelled
      10 |       |
      11 |       |
          +-------+
```

When we have a pointer to the entire struct, the pointer points to the first byte of the struct - so the `x` field is right there, the `y` field is one byte further on, etc. To simplify your code, you should access the fields by using the pointer-to-struct as the base pointer, and then use constant offsets to access the fields. For instance, if the pointer to the struct is in `$s1` and you want to read the 'name' pointer, you would do so as follows:

```
lw   $t0, 4($s1)
```

In this project, I will call various functions that you have implemented; for most, I will pass a pointer to this struct (along with other parameters).

2.1 Required Filenames to Turn in

Name your assembly language file `asm4.s`.

2.2 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub, addu, addiu`
- `and, andi, or, ori, xor, xori, nor`
- `beq, bne, j`

- `jal, jr`
- `slt, slti`
- `sll, sra, srl`
- `lw, lh, lb, sw, sh, sb`
- `la`
- `syscall`
- `mult, div, mfhi, mflo`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), **do not use them!** We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

2.3 Properly Saving Registers

The testcase includes lots of features which are designed to verify that you are saving registers properly.

First of all, it initializes all of the `sX` registers - and `$fp` as well - to various 32-bit values. At the end of the testcase, we'll print out these values (along with a word we passed onto the stack); if you have saved those registers properly - and also restored `$sp` to the proper location - then your output will match the expected output.

On the other side of things you must know that if you decide to use `tX` registers in your code (and sometimes, that's a great strategy), then you **must not** assume that they are unchanged by a function call.

To test that, every function in the testcases which you might call will **intentionally** corrupt every `tX` register, `aX` register, and `vX` register. (Many will return a value in `v0`; in that case, they will only corrupt `v1`.)

Thus, if your code depends on any value that you've stored in a `tX` register - and you don't save it properly - then your program will operate improperly.

2.4 Matching the Output

You must match the expected output **exactly,** byte for byte. Every task ends with a blank line (if it does anything at all); do not print the blank line if you do not perform the task. (Thus, if a testcase asks you to perform no tasks, your code will print nothing at all.)

To find exactly the correct spelling, spacing, and other details, always look at the `.out` file for each example testcase I've provided. **Any example (or stated requirement) in this spec is approximate; if it doesn't match the `.out` file, then trust the `.out` file.**

(Of course, if there are any cases in the spec which are not covered by any testcase that I've provided, then use the spec as the authoritative source.)

3 Tasks

Your file must define the functions listed below:

3.1 `void turtle_init(Turtle *obj, char *name)`

This is the "constructor" for the Turtle object (or, if you prefer, the "Turtle class"). I have already allocated space for the object, and I will pass you a pointer to it. I will also pass you a pointer to the string.

You must initialize all of the fields of the struct; it should start at the position $(0, 0)$ and direction 0 (North). (Actually set each of the fields - I may pass you an object which is full of trash!)

WARNING: There will be multiple `Turtle` objects! So be sure to actually store the information inside the object - do not attempt to use any global variables!

In fact, in this program, all global variables are banned - except for constants. Remember, strings are constants, and that's fine. You are also allowed to have integer constants - these are just like global variables, except that you never modify them.

3.2 `void turtle_debug(Turtle *obj)`

I'll call this function to make sure that you set the fields of your turtles correctly. Print out all of the fields of the `Turtle`, as follows:

```
Turtle "charlie"
  pos 10,3
  dir North
  odometer 1234
```

(Add a blank line after printing the above text.) The lines after the first line are two spaces indented.

NOTE: I will also sometimes print out the contents of the struct as raw bytes, instead of calling this function. Thus, it's important that you actually use the correct bytes for the various fields of the struct.

3.3 `void turtle_turnLeft(Turtle *obj)`, `void turtle_turnRight(Turtle *obj)`

These functions turn the turtle 90 degress to the left or right. Update the `dir` field.

Make sure to limit the range to $[0, 3]$ **(inclusive).** That is, if the turtle is facing North (`dir=0`) and turns left, the correct new `dir` is 3, **not** -1.

You can accomplish with conditional branches, of course. But you can do it even more easily with some bitwise operations!

3.4 `turtle_move(Turtle *obj, int dist)`

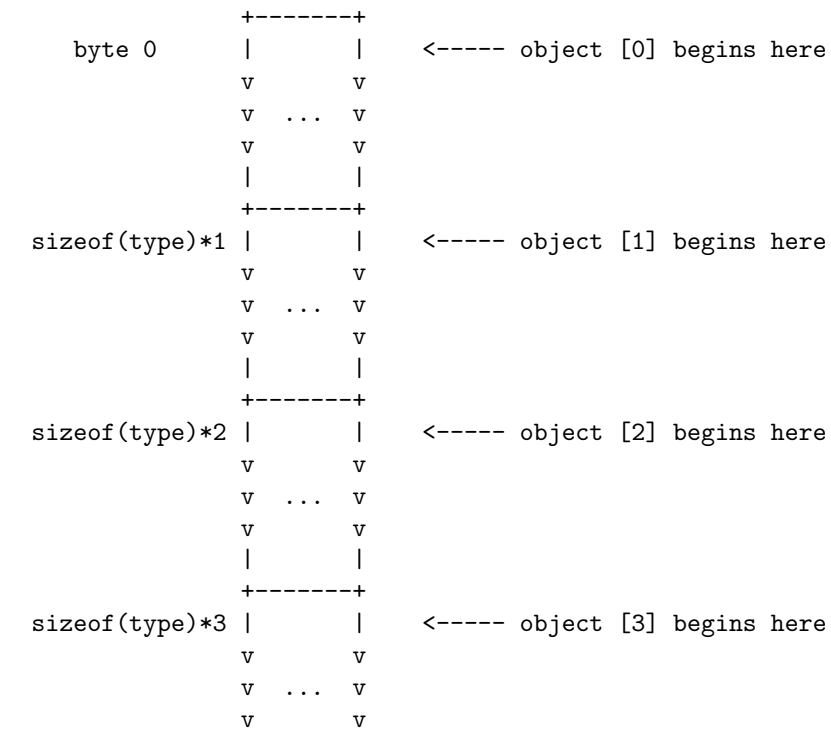
This function moves the turtle forward by a certain distance. You may assume that the distance is valid; it might be positive, negative, or zero. If the distance is positive, then it moves in the direction that it is facing; if it is negative, then it moves in the opposite direction (but keeps its current facing).

You must clamp the x and y coordinates after every move; the turtle must not move off of the "board." For both coordinates, the minimum allowed value is -10, and the maximum is 10.

You must also update the `odometer` field; increment it by the **absolute value** of the distance travelled. Do **not** clamp this value. For instance, if the function asks to move forward by 50, the turtle will move in that direction until it hits the edge of the board (and thus it will move far less than 50). But the `odometer` must increase by exactly 50.

3.5 `int turtle_searchName(Turtle *arr, int arrLen, char *needle)`

This function searches through an **array** of Turtle objects. In an array of structs, the objects are arranged in the array just like fields in a giant meta-struct:



In this function, you will search through the various `Turtle` objects in the array, looking for one which has a name which matches the `needle` parameter. If you find one, return the array index where you found it. (Don't worry about duplicates; simply return the first you find.)

If you can't find any object matching the name, return -1.

To make this easier for you, I have provided an implementation of `strcmp()` (the function from the C standard library) for you. I **encourage** you to call it to perform the string comparisons.

3.6 `void turtle_sortByX_indirect(Turtle **arr, int arrLen)`

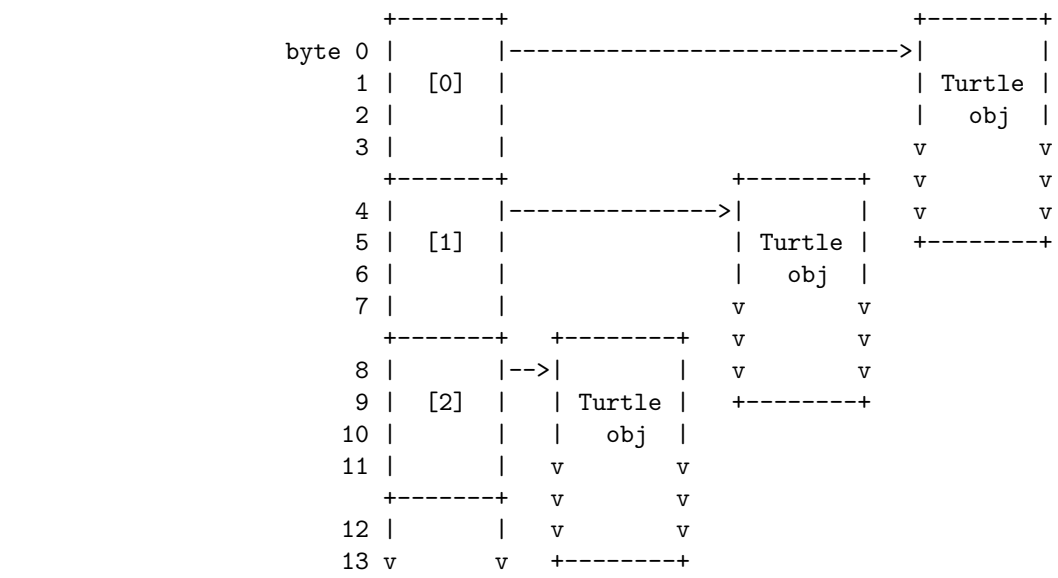
This function requires that you sort a set of `Turtle` objects. Sort them by their current `x` values. (I'll make sure that no two objects have the same `x` value.) While you are free to use an advanced sort if you want, I **encourage** you to use Bubble Sort: it's simple, and easy to implement.

When you decide to swap two of the objects, do **not** attempt to actually modify the objects themselves! Instead, simply swap the pointers inside the array! Not only is this better for performance, it is good object-oriented practice.

3.6.1 How the Array is Structured

This function gives you a set of `Turtle` objects, but this time, it does not arrange them as an array. Instead, it gives you an **array of pointers.** (Notice that the `arr` parameter is a pointer-pointer. That means that is a pointer to a memory location, and that other memory location is a pointer. In this case, we are interpreting the parameter as pointer-to-first-element-of-an-array-of-pointers.)

An array of pointers looks like this:



That is, the array is made up of a bunch of 4-byte values, which are pointers; the pointers give you the address of `Turtle` objects which are scattered all over memory. (Do **not** assume that all of the `Turtle`s are next to each other!)

(BTW, this is how Java handles arrays of objects. Java does **not** allow you to actually put objects next to each other; it always requires that you have arrays of references, to objects stored elsewhere. C is cool because, like assembly, it gives you the flexibility to use either strategy.)

4 Running Your Code

You should always run your code using the grading script before you turn it in. However, while you are writing (or debugging) your code, it is often handy to run the code yourself.

4.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from the Mars website. You may be able to just double-click it in your operating system; if not, then you can run it by typing the following command:

```
java -jar <marsJarFileName>
```

This will open a GUI, where you can edit and then run your code. Put your code, plus **one**[1] testcase, in some directory. Open your code in the Mars editor; you can edit it there. When it's ready to run, assemble it (F3), run it (F5), or step through it one instruction at a time (F7). You can even step **backwards** in time (F8)!

4.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the `Settings` menu, and set two options:

- `Assemble all files in directory` - so your code will find, and link with, the testcase
- `Initialize Program Counter to 'main' if defined` - so that the program will begin with `main()` (in the testcase) instead of the first line of code in your file.

4.2 Running Mars at the Command Line

You can also run Mars without a GUI. This will only print out the things that you explicitly print inside your program (and errors, of course).[2] However, it's an easy way to test simple fixes. (And of course, it's how the grading script works.) Perhaps the nicest part of it is that (unlike the GUI, as far as I can tell), you can tell Mars exactly what files you want to run - so multiple testcases in the directory is OK.

To run Mars at the command line, type the following command:

[1] Why can't you put multiple testcases in the directory at the same time? As far as I can tell (though I'm just learning Mars myself), the Mars GUI only runs in two modes: either (a) it runs only one file, or (b) it runs **all** of the files in the same directory. If you put multiple testcases in the directory, it will get duplicate-symbol errors.

[2] Mars has lots of additional options that allow you to dump more information, but I haven't investigated them. If you find something useful, be sure to share it with the class!

```
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s
```

5 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines, misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

5.1 `mips_checker.pl`

In addition to downloading `grade_asm4`, you should also download `mips_checker.pl`, and put it in the same directory. The grading script will call the checker script.

5.2 Testcases

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

5.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm4`, along with a helper script, `mips_checker.pl`. Place both scripts, all of the testcase files (including their `.out` files), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac or Linux box, but no promises!)

5.4 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share your testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

6 Turning in Your Solution

You must turn in your code using Gradescope. Turn in only your program; do not turn in any testcases or other files.