CS 211 Lab 2

Control Statements, Functions and Structures

Winter 2021

Today we are going to practice navigating in the shell and writing basic C code.

Getting Started

Let's get started by logging into a remote Northwestern workstation. If you need help remember the steps, review Lab 1.

The list of remote Northwestern workstations can be found here: it.eecs.northwestern.edu/info/2015/11/03/info-labs.html#workstations

Enabling the CS 211 build environment

Type the command `% exec fish` into the shell to execute *fish* in place of your current shell.

(If your shell complains that *fish* isn't found, that most likely means you missed the step in Lab 1 where you were supposed to run `% ~cs211/setup211`. You can do it now and then run `% exec tcsh` once to make the change affect your current login session.)

If you login using SSH on the command line then you can run `% ssh` ⟨*user*⟩`@`⟨*host*⟩ `-t fish` to have it run *fish* immediately upon login.

Getting the code

Recall our basic Unix commands: *cd*, *ls*, *mkdir*, and *pwd*. What do they stand for and what do they do? If you don't remember, try reading their manual pages.

Run `% man ls` or `% man pwd`.

We suggest that organize your home directory by keeping your CS 211 files in subdirectory named cs211/, but it's up to you. If you have such a directory, change into it and then extract the tarball for this lab:

```
% cd cs211
% tar -xkvf ~cs211/lab/lab02.tgz
```

You should now have a directory called lab02.

Note that the directories cs211/ and ~cs211/ do not mean the same thing. The former means a subdirectory of the current directory named "cs211," whereas the latter means the home directory for a user named "cs211." When ~ is written by itself, it means the *current user's* home directory. Given all that, what do you think ~/cs211/ means?

Writing the code

Navigate into to your lab02 directory using *cd*, and open up src/sum.c in Emacs using

```
% emacs src/sum.c
```

Notice that there is already some skeletons of functions and some code in *main*() here.

Iteration

First, find the function called *sum_numbers*(). We are going to use this function to sum up all of the numbers from 1 to `num`. If you remember from class, we have a few ways of iterating through numbers, most notably **`for`** and **`while`**. We will be using both, but first we will be using **`while`**.

Notice that *sum_numbers*() is going to return an **`int`**.

`while` *loops*

As we learned in class, a **`while`** loop has the following syntax:

```
while (⟨test-expression⟩) {
    // Repeats the body statements until the test
    // expression evaluates to `false`:
    ⟨body-statements⟩
}
```

Note that in while loops we usually will use a Boolean expression for ⟨*test-expression*⟩ (an expression which evaluates to ‘true‘ or ‘false‘)

Use a while loop inside our *sum_numbers*() in order to add the numbers from 1 to `num` together. Make sure to use a **`return`** statement to return the sum that we aggregated!

Remember that we have the ++ and += operators if we need them.

Once you think that your function works as intended, save and exit emacs. If you remember from last week, we used the `make` command in order to turn our C file into machine code. Run:

C-x C-s to save and C-x C-c to exit

```
% make sum
```

Remember, make works as follows: `% make [target]`. Target is the name of the executable file that will be built by the make command.

If everything works, if we list our files, we should now see a file called `lab`. To run it, the command is:

```
% ./sum
```

See if your value looks right! If it doesn’t, don’t worry. These labs are designed so you can practice. Investigate and try to see what went wrong. Play around with the value of `num` and see how it affects the result.

Error messages may look scary, but in reality, they’re there to help you! Not intimidate you!

`for` *loops*

Once we have everything working with our **`while`** loop, let’s work on using a **`for`** loop. To help you remember the syntax, here is an example of a **`for`** loop to print out the values from 1 to 5:

```
for (size_t i = 1; i <= 5; ++i) {
    // Note that the code inside the curly braces is the code
    // that is executed for each iteration of the `for` loop.
    printf("%zu\n", i);
}
```

Go back to our *sum_numbers*() function, and try to replace your `while` loop expression with a `for` loop to accomplish the same task.

Once you are done, make and run your file. See if everything looks the same! If not, no worries, go back and try again!

Structs

Structs are an important tool in C for grouping data together. In your structs.c you will notice that we created an `apple` structure for you. This is so that we can organize attributes of `struct apple`s together in a convenient way. If you look inside our `apple` struct, we decided that we will want to know the weight, the variety, and the color of our apples. In our *main*(), we created an example of a Red Delicious apple. Now, create your own type of apple (you'll need to define it as a type `struct apple`), and give it those three attributes. Add in a print statement (we gave you an example one) to print out your new `struct apple`. Make and run your *structs* program! Hopefully everything works as expected! If not, don't fret or get upset, go back and make changes!

How 'bout 'dem apples?

Now that we know all about `struct apple`s, let's create our own structure. Define a structure of your favorite animal, and give your animal three attributes, with one of them being age. Don't forget to give your attributes types. You can create a new struct that looks very similar to the `struct apple` we created.

Don't forget the semicolon after the closing brace.

Once you have created your animal, go into *main*() and create an instance of your animal, assign it those three attributes, and then create a print statement to print out information about your animal. These print statements are getting annoying; we'll tackle that soon.

Make and run *structs*, and see if your new animal shows up the way that you intended. Hopefully everything works! If not, as usual, go back and try and find what went wrong and update your code.

Creating your own function

So far, we've been filling in skeleton functions that were provided for you. Now it's time to write your own function from scratch. Remember how annoying it was to type out the `printf` lines each time you wanted to print out your animal? We're going to abstract that out and replace it with a simple call to a function!

The "(3)" after "*printf*" gives the section in the Unix manual that contains the documentation for the *printf*() function, which means you can look it up with the command `% man 3 printf`. If you omit the section number 3 and run just `% man printf`, you get documentation for the shell's `printf` command rather than the C library's *printf*() function.

Write a function called *print_animal*() that uses *printf*(3) to print out your animal's three attributes. Note that this should take in one argument (of the same type as your animal struct). Think about what type your function should return!

Note that the void return type signifies that nothing is returned.

Once you wrote your function, go to your *main*() function and replace your print statement with a call to *print_animal*().

Remember to pass your animal instance to the function.

Make and run *structs*, and see if everything still works!

Control statements

Now that we have gotten the hang of structs and functions, let's practice our control statements. Go back to our *print_animal*() function. Remember from class that **`if`** statements have the following basic syntax:

```
if (⟨test-expression⟩) {
    // Do these if the test expression is `true`:
    ⟨then-statements⟩
} else {
    // Do these if the test expression is `false`:
    ⟨else-statements⟩
    // (The `else` clause is optional.)
}
```

Using an **`if`**–**`else`** statement, check your animals age and add to your *print_animal*() function a line to print out "This animal is old!" if the animal is at least 10 and print out "This animal is not that old" if the animal is younger than 10.

Build and run *structs*, and see if this feature is working.