Lab 2: An Introduction to Flow Control in C

CSE/IT 113L

NMT Department of Computer Science and Engineering

"There's a good part of Computer Science that's like magic. Unfortunately there's a bad part of Computer Science that's like religion."

— Hal Abelson

"There are only 2 hard problems in computer science: cache invalidation and naming things."

— Phil Karlton

"... the ingredients that have to be in place in order to get an ah hah moment. You have to be working on a problem, but also have to be able to have "off time" so that the brain can work on the back burner."

— Barbara Liskov

Contents

1 Introduction

In Lab 1, you created a program which read a variety of input from the user. The numbers were used to calculate the results of various geometric operations and the results were displayed to the screen (stdin). Unlike a real calculator, the user could not choose which calculation to perform. In this lab, you will learn how to have the user choose the desired calculation.

Lab 2 continues usage of the `if` statement and the `switch` statement. These statements are called conditional execution because they allow us to choose one action from one or more possible actions.

Lab 2 also utilizes the concept of reading character input from the keyboard and performing comparisons on character values.

2 Overview

In order to receive full credit for this lab, you must **write a simple geometry calculator program**. The program will ask the user what calculation to perform, ask for input, and then carry out the desired calculation. If there is an error in user input at any step along the way, the program will print a message and exit the program.

2.1 Header Files

You will create three files for your new geometric calculator program: `lab2.c`,`functions.h`, and `functions.c`. The `*.c` files are source code files, and the `*.h` file is known as a header file. Header files contain function prototypes, `#define` statements and other preprocessor directives.

You will split your code across multiple files in the following way:

1. `lab2.c` will contain only the main() function of your program. That is the logic of your program. At the top of the file, you will need to include the `functions.h` file. This tells the compiler to add the function prototypes from `functions.h`
2. `functions.h` contains the function prototypes of the functions you will implement.
3. The definition of the functions are in `functions.c`. Note that `functions.c` should include `functions.h`, this is so you can put preprocessor directives in `functions.h` and they can be used in `functions.c`

A couple of things to note about the files you create for the geometric calculator. They should include more Doxygen annotations (@file, etc) at the top of the file. **Every C file you turn in from now on needs to have this comment block with the tags filled out at the top of file**. Make sure you fill these tags out with the correct information.

The header file `functions.h` should contain what is known as an include guard, which prevents multiple copies of the file being processed by the compiler. In `functions.h`, if the variable `FUNCTIONS_H` is undefined the preprocessor will continue processing the file `functions.h`. The first time the file is processed the variable `FUNCTIONS_H` is defined preventing further reads of the file by other files that are being compiled.

2.2 Compiling Programs with Multiple Files

When your program is split across multiple files, you have to compile the program in a different manner. The first step is to create an *object* file for every C source file (`*.c`) that does not have a main function.

```
$ gcc -g -Wall -c functions.c
```

The above command creates the file `functions.o`. You can see this by doing a `ls` in the current directory.

Once you have created all the object files your main() function uses, the next step is to link in all the object files (`*.o`) to the file with the main() function.

```
$ gcc -g -Wall lab2.c functions.o -o lab2
```

The above command creates the executable file named `lab2`, which you can run with the following command:

```
$ ./lab2
```

2.3 Characters and Comparisons

Up to this point you have dealt with two data types: `int` for integers and `double` for floating-point numbers. This lab introduces a third type `char`, which holds a single character (one byte) such as `A, B, C, a, b, c` etc. To assign a character to a variable we use *single* quotes. For example,

```
1 char c = 'a';
2 printf("%c\n", c); /* what does this print -- try it*/
```

Note that we are using *single* quotes. The following is wrong

```
char c = "a"; /* this is not correct */
```

Double quotes are used to indicate **strings** (you will learn exactly what a string is later in the course). For right now, recognize that `'a'` is not the same as `"a"` and single quotes are used for **character** data types.

Also make sure you understand the difference between numbers and their character representation. For example, `'5'` is not the same as `5`. `'5'` is a character type and has an ASCII value of `0x35 or in decimal 53`; `5` is an integer type and represents the number `5`. Remember like UNIX, character types are case-sensitive in C: `'a'` is not the same as `'A'`.

If you want to learn more about ASCII characters you can type **man ascii** in your terminal or look online, such as at this website: `www.asciitable.com`

To compare character types you use the equality operator (==)

```
char c = 'd';
char d = 'd';

if (c == d) {
        printf("c is d\n");
} else {
        printf("c is not d\n");
}
```

What does the above code fragment print? Try it.

2.4 getchar()

Rather than using `scanf()` for all user input you will be using `getchar()` for character input and `scanf()` for integer and double input.

As you saw during the lecture, `getchar()` is preferred over `scanf()` for character input. For example:

```
int c;
int tmp;
printf("Please enter a character: ");

while ((tmp = getchar()) != '\n')
    c = tmp;

printf("c = %c\n", c); /* %c token is for characters */
```

This will print the last character entered before the Enter key is pressed. Try it.

3 Lab Requirements

The following subsections describe what you will write and submit for this lab. **Check that you have met all of the requirements for this lab before you submit it for grading.**

3.1 Rock-Paper-Scissors-Lizard-Spock

Exercise 1 (rock-spock.c, rs.script).
The tarball contains a file named `rock-spock.c`. The file contains source code for the game rock-paper-scissors-lizard-Spock. Unfortunately, the game is not finished. Your job is to finish the program. First run the code to get a feel for what it does. Then add code to do the following in `rock-spock.c`:

1. Check to make sure the player enters a correct move.
2. Do not use `scanf()` for input, only use `getchar()`!
3. Use `switch` statements to determine the winner of Rock-paper-scissors-lizard-Spock. See `http://en.wikipedia.org/wiki/Rock-paper-scissors-lizard-Spock` for the rules of the game.
4. Implement the function `winner()`. The prototype is given. Do not change it. This function returns the winner. It should be called from `main()`.
5. Print out the winner. Implement the function `print_winner()`. The prototype is given. Do not change it. Use the provided phrases. It should be called from `main()` and use as input the value returned from `winner()`.
6. Do not add any other functions to the code. You don't need any more.
7. Implement the while loop to play games until `'Q'` is entered.
8. Capture the output of at least 10 plays in a script file named `rs.script`

3.2 Geometry Calculator

Exercise 2 (lab2.c, l2.script, functions.c, functions.h).
Your program ***must*** be able to execute all the geometry calculations from Lab 1.

You ***must***

- use a nested `switch` statement to determine which calculation to perform. You will first ask what general type of calculation (Circle, Regular Polygon, Rectangle, or Triangle) the user wants. And based on that input, determine which operation to perform, get user input to perform the calculation, and carry out the calculation.
- error check user input at every step of the way. You should accept both upper and lower case values for menu options, and since this is geometry the values the user entered should be greater or equal to zero. If an error occurs, print a message to the screen and exit the program. Depending on the error, you will either handle it with a `switch` or an `if` statement. See the sample code (below) for an example.
- use `getchar()` for character input.
- use `scanf()` for integer and double input.

3.2.1 Sample Code - Nested Switch Statements

The following program gives an example of nested switch statements.

```
char in()
{
  char c;
  while ((tmp = getchar()) != '\n')
    c = tmp;

  return c;
}

int main()
{
  printf("What position did you start the race in? (1-3) ");
  char start = in();
  printf("What position did you end the race in? (1-3) ");
  char end = in();

  switch(start) {
  case '1':
    switch(end) {
    case '1':
      printf("You stayed in the same position!\n");
      // without break, the flow of code can become unpredictable
      // this breaks out of the switch(end) statement
      break;
    // default runs for everything that isn't listed above
    // think of it as "else"
    default:
      printf("You went down the leaderboard.\n");
    }
    // this breaks out of the switch(start) statement
    break;
```

```
32
33    case '2':
34      switch(end) {
35      case '1':
36        printf("You improved!\n");
37        break;
38      case '2':
39        printf("You stayed in the same position!\n");
40        break;
41      case '3':
42        printf("You went down the leaderboard.\n");
43        break;
44      }
45      break;
46
47    case '3':
48      switch(end) {
49      case '3':
50        printf("You stayed in the same position!\n");
51        break;
52      default:
53        printf("You improved!\n");
54      }
55      break;
56    }
57
58    return 0;
59  }
```

3.2.2 Sample Output

The output of your program will look as follows.

Important: The output must be calculated by your program. That is the point of this program. Do *not* print the exact output as given here as an example of the program in operation. Your program should perform the user-specified arithmetic operation based on the numerical values and operator read in from the keyboard.

```
Welcome to Wile E. Coyote's Geometry Calculator!
Guaranteed to Pythagorize the Roadrunner in his tracks!

Please select a geometry calculation:
C. Circles
P. Regular Polygons
R. Rectangles
T. Right Triangles

Please enter your choice (C, P, R, T): C
A. Area of a circle
```

```
C. Circumference of a circle
Please enter your choice (A, C): A
Enter the radius of the circle: 2
The area of a circle with radius 2.0 is 12.566371
```

or if the user made an error in the value of the radius entered:

```
Welcome to Wile E. Coyote's Geometry Calculator!
Guaranteed to Pythagorize the Roadrunner in his tracks!

Please select a geometry calculation:
C. Circles
P. Regular Polygons
R. Rectangles
T. Right Triangles

Please enter your choice (C, P, R, T): C
A. Area of a circle
C. Circumference of a circle
Please enter your choice (A, C): A
Enter the radius of the circle: -2
Error: radius has to be greater than or equal to zero
Goodbye.
```

3.3 Additional Problem

Exercise 3 (pp8.c, cpama.script).
In addition to the the above requirements, turn in the following problem from Chapter 5 of *C programming: A Modern Approach*: Programming Project #8.

The following table shows the daily flights from one city to another:

Departure Time	*Arrival Time*
8:00 am	10:16 am
9:43 am	11:52 am
11:10 am	1:31 pm
12:47 pm	3:00 pm
2:00 pm	4:08 pm
3:45 pm	5:55 pm
7:00 pm	9:20 pm
9:45 pm	11:58 pm

Write a program that asks the user to enter a time (expressed in hours and minutes, using a 24-hour clock). You program should then display the departure and arrival times for

the flight whose departure is closest to that entered by the user:

```
Enter a 24-hour time in hours and minutes (HH MM): 13 15
Closest departure time is 12:47 pm, arriving at 3:00 pm.
```

Note Bene: Convert the input into a time expressed in minutes since midnight, and compare it to the departure times, also expressed in minutes since midnight. For example, 13:15 is $13x60 + 15 = 795$ minutes since midnight, which is closer to 12:47 pm (767 minutes since midnight) than to any of the other departure times.

Name this file `pp8.c`. Generate a script file of you compiling and running your `pp8.c` file and name it `cpama.script`.

Make sure your code for the additional problems follows the coding style of the class and is commented.

4 README

Exercise 4 (README).
Every lab you turn in will include a README file. The name of the file is README with no file extension.The README will always include the following two sections *Purpose* and *Conclusion*:

- **Purpose:** describe what the program does (i.e. what problem it solves). Keep this brief.
- **Conclusion:**
  - What did you learn. What new aspect of programming did you learn in this lab? This is the portion of the lab where you want to be analytical about what you learned.
  - Did the Pair Programming prelab help you in solving this solo lab?
  - Did you encounter any problems? How did you fix those problems?
  - Did your code still contain bugs in your final submission? If so, list those bugs.
  - What improvements might you make? In other words, if you were to refactor your lab code, what would you update?

The conclusion does not have to be lengthy, but it should be thorough.

Some labs will require a special **Pseudo-code** section. If a lab requires a pseudocode section, it will be explicitly mentioned in that lab.

Submitting

You should submit your code as a tarball file that contains all the exercise files for this lab. The submitted file should be named (**note the lowercase**)

`cse113_firstname_lastname_lab2.tar.gz`

If you require a refresher on generating a tar archive file, instructions can be found in Section 6 - Creating Tar archives of Prelab 0.

Upload your `.tar.gz` file to Canvas.

List of Files to Submit

Exercises start on page 4.