Prelab 2: An Introduction to Flow Control in C

CSE/IT 113L

NMT Department of Computer Science and Engineering

"C++ and Java, say, are presumably growing faster than plain C, but I bet C will still be around."

— Dennis Ritchie

"Persistence is very important. You should not give up unless you are forced to give up."

— Elon Musk

"I'd just like to interject for a moment. What you're referring to as Linux, is in fact, GNU/Linux, or as I've recently taken to calling it, GNU plus Linux."

— Richard Stallman

Contents

1 Introduction

In Lab 1, you created a geometry calculator which read in a variety of numbers from the user and performed some geometric calculations. Unlike a regular calculator, the user had no choice of which operation was performed. In Lab 2, you will allow the user to specify the desired geometric calculation. This will allow you to create programs that respond to different user input.

Lab 2 will introduce the `if` statement and the `switch` statement. These statements are called branching (or decision) structures because, as flow control structures, they allow us to choose one action from one or more possible actions.

In addition to branching structures, Lab 2 also introduces the concept of reading character input from the keyboard and performing comparisons on character values.

2 Overview

In order to complete Lab 2, you will need the following:

- An understanding of how to compile programs (Lab 0)
- An understanding of how to read user input (Lab 1)

In order to complete this prelab, read the material in this section and answer the questions in Section 3.2.

2.1 Material

In Lab 2, you will be altering the geometry calculator code you created in Lab 1 in order to allow the calculator's user to choose a geometric operation of choice. In order to allow user choice, you need to implement flow control. *Flow control* is simply a means to control which parts of your program run depending on if certain conditions (criteria) are met or not. In particular, you will be implementing an `if` statement and a `switch` statement, using conditionals and logical expressions.

2.2 Conditionals

Conditionals are integral parts of programming, because they are the cornerstone to implementing *flow control*. A conditional is simply a decision point in your code and is implemented using a logical (Boolean) or relational expression or a combination of the two. A logical or relational expression is a statement that evaluates to either true or false. In C, there is not a Boolean data type: true values are represented by any integer not equal to `0` (typically 1), a false value is equal to `0`.

2.2.1 Logical and Relational operators

A few of the logical and relational operators that you will often use are:

Expression	Meaning
`x`	If x is 0, false; otherwise, true.
`!x`	If x is 0, true; otherwise, false.
`x == y`	True if x and y are equivalent.
`x != y`	True if x and y are not equivalent.
`x < y`	True if x is less than y.
`x > y`	True if x is greater than y.
`x <= y`	True if x is less than or equal to y.
`x >= y`	True if x is greater than or equal to y.

A common mistake (and often subtle run time error) is to type `x = y` when you mean `x == y`. As both are valid C statements the compiler will not complain! In `gcc` if you compile with the `-Wall` flag, the compiler will produce the warning:

```
warning: suggest parentheses around assignment used as truth value
 ↪  [-Wparentheses]
```

Here is a simple C program to try. Initially the variable `y` is `0`. Run the code as is, then change `y` to `4` and run, and then change `y` to `-4` and run. Compile with and without the `-Wall` flag. What happens?

```c
#include <stdio.h>

int main(void)
{
        int x = 3;
        int y = 0;

        if (x = y) {
                /* the body of the if statement is executed
                   if (x > 0) or (x < 0) */
                printf("if: x = %d\n", x);
        } else {
               /* the body of the else statement is executed
                    if (x = 0) */
                printf("else: x = %d\n", x);
        }

        return 0;
}
```

Remember:

- == means "compare for equality."
- = means "assign the value on the right hand side to the variable on the left hand side."

There are many other logical and relational operators available, and we will discuss some of them in a later lab.

2.2.2 Compound statements

In addition to the basic operators, you can use logical operators to build compound statements. Three such logical operators are *NOT*, *AND* and *OR*. The logical *NOT* is represented by `!` in C. The logical *AND* operator is represented by `&&` in C. The logical *OR* is represented by `||` in C.

Expression	Meaning
`!(condition)`	Negate the value of the condition; inverts the meaning of the condition.
`(condition 1) && (condition 2)`	True if condition 1 ***and*** condition 2 are both true. *Short circuit*: condition 2 is not attempted if condition 1 is false. Why?
`(condition 1) || (condition 2)`	True if either condition 1 is true ***or*** condition 2 is true ***or*** both condition 1 and condition 2 are both true. *Short circuit*: condition 2 is not attempted if condition 1 is true. Why?

For example, `((x < y) && (x < z))` will evaluate to true if `(x < y)` is true *and* `(x < z)` is true.

C will ***short circuit*** the evaluation of complex expressions to speed up program execution. If you have a complex **and** expression, once the first condition is deemed false, there is no need to evaluate the second condition as the entire expression is false. To save CPU cycles, C will not evaluate the rest of the conditionals. This is known as *short-circuiting*. Short-circuiting can lead to subtle run time errors if you use assignment operations in your conditionals.

2.3 The `if` statement

The `if` statement allows you, at runtime, to decide whether or not a statement or a collection of statements will be executed. For example, consider the following code segment:

```
1 if (x == 3) {
2     printf("x = 3\n");
3 }
```

In the above `if` statement, the `printf()` function will be executed if and only if, `x` is equivalent to `3`. In its most basic form, the `if` statement has the following structure:

```
if (condition) {
     /* statements to execute when condition is true */
}
```

Other forms of the `if`-statement use `else` or `else-if` blocks:

```
if (condition) {
     /* statements to execute when condition is true */
} else {
     /* statements to execute when condition is false */
}
```

```
if (condition) {
     /* statements to execute when the if condition is true */
} else if (condition) {
     /* statements to execute when the else if condition is true */
} else {
     /* statements to execute when the else condition is true
     That is, when all previous if and else if conditions are false. */
}
```

Note: You can have as many `else if` blocks as needed and in all cases the `else` block is not strictly necessary.

2.3.1 Blocks

You have probably noticed that the `if` statement is framed by curly braces, `{}`. The braces indicate the body of the `if` statement and indicate the block of code to run when the condition is true. Likewise the curly braces indicate the block of code to run when the condition is true for `else if` or `else`.

The braces following `if` statements are not necessary if you only need to execute ***one*** statement:

```
if (condition)
    /* statement to run if true */
```

By framing the `if`, `else if`, `else` statements in curly braces, you can execute as many statements as you need, because the whole block of code within each set of curly braces is executed as a unit.

2.3.2 Nesting

You can **nest** `if` statements inside of other `if` statements in order to perform more complex evaluation tasks:

```
if (foo == 1) {
     /* todo */
} else {
     if (foo == 2) {
          /* todo */
     } else {
          if (foo == 3) {
                  /* todo */
          } else {
                 /* todo */
          }
     }
}
```

You can continue on in this manner, but this is an example of *bad coding style* as you are indenting needlessly. The condition involves a test of the same variable. In this case, it is wiser to simplify the structure by using an `if/else if` waterfall structure:

```
if (foo == 1) {
     /* todo */
} else if (foo == 2) {
     /* todo */
} else if (foo == 3) {
     /* todo */
} else {
     /* none of the above is true; default case */
}
```

Another use for nesting is when your problem requires further processing on a *different* variable or variables after entering an initial selection control flow. For example, take a look at the nested selection statements below:

```
if (foo == 1) {
        if (bar < foo) {
                if (foobar == foo * bar)
                        foo = foobar;
        }
} else if (foo == 2) {
        if (bar == 19) {
             /* todo */
        } else if (bar == 66) {
```

```
            /*todo */
        }
} else if (foo == 3) {
        if (bar > foo) {
            /* todo */
        } else if (bar < foo) {
            /*todo */
        } else {
            /*todo */
        }
} else {
     /* none of the above is true; default case */
}
```

2.4 The `switch` statement

Nested `if` statements are a fairly standard way to represent a complex set of choices. However, sometimes you want to evaluate a set of alternatives that don't require the full power of Boolean expressions. For situations that require only a test of equivalence of integral (character or integer) data-types you can use the `switch` statement. The following example is the previous nested `if`/`else` code translated into a `switch` statement:

```
switch (foo) {
case 1: /* equivalent to test (foo == 1)  */
        /* todo -- executes if foo == 1*/
        break; /* execution moves to the '}'
                  at the end of the switch*/
case 2:
        /* todo */
        break;
case 3:
        /* todo */
        break;
default:
        /* none of the above is true; default case */
        break;
}
```

The `switch` statement is easier to use, but remember: you can only use it when you are doing a simple test for equivalence. The following are guidelines on effectively using a `switch` statement:

- Put a `break;` at the end of each case. If you don't, the program will continue executing statements until it either encounters a `break` or reaches the end of the `switch` statement. This is known as ***fall through***.
- You can put as many `case` statements as you want before the code you want to execute if the equivalence is true. Each case will execute the same code. For example, if you wanted to do

the same thing for odd digits less than 10 and something different for even digits less than 10 you could program the following `switch` statement:

```
switch (foo) {
case 1:
case 3:
case 5:
case 7:
case 9:
        /* todo */
        break;
case 2:
case 4:
case 6:
case 8:
        /* todo */
        break;
}
```

- Each `case` label is followed by a colon (:), not a semi-colon.
- You do not need a default case.
- `switch` statements only work for ***integral*** data-types. That is, the data-type you are switching on must be an `int` or a `char` data-type.

2.5 The `while` loop

Prelab 3 will have you programming while loops, but we want to introduce them to you now so you understand the code in the following section. A while loop is the simplest and most fundamental loop statement. It has the form:

$$while(expression)statement$$

The expression inside the parentheses is the **controlling expression** and the statement after the parentheses is the while **loop body**. An example of a while loop would be:

```
while (i < n)     /* controlling expression */
    i = i * 2;    /* loop body */
```

This statement computes the smallest power of 2 that is greater than or equal to a number n.

2.6 Reading and storing character input

In your previous lab, you only needed to store numbers. While numbers are useful, sometimes you want a little *character* in your code ...

`scanf()` works well for numeric input, but it is not particularly useful when it comes to character data. Compile and run this simple program:

```c
#include <stdio.h>

int main()
{
        char a;
        char b;

        printf("enter a char: ");
        scanf("%c", &a);

        printf("enter another char: ");
        scanf("%c", &b);

        printf("\nb = %d\n", b);

        return 0;
}
```

When you run the program, you will notice that the first `scanf()` works but the second doesn't. Why is that? When you entered the first character, you followed it with a newline character. `scanf()` stores the newline character and when the second `scanf()` is called it already has a character value – the newline. So it just uses that. You can confirm this as `b = 10` will be printed. 10 is the ASCII value for the newline.

A better choice to use for character input from the terminal or standard input (stdin) is the function `getchar()` whose function prototype is found in the standard header file `stdio.h`. For technical reasons, `getchar()` returns an `int`, not a `char`.

If you want to compare a variable to a character you use **single quotes** to delimit the character. It is important to understand the difference between single and double quotes in C as they indicate different data types. Single quotes indicate a `char` type; double quotes indicate a `char*` (or string) type - technically an array of characters that are NULL terminated. Unlike Python, `C` does **not** treat these two data types the same!

The following short program demonstrates a `while` loop, `getchar()`, and a `switch` statement using character comparison:

```c
#include <stdio.h>

int main()
{

     int answer;
     int tmp;

     printf("Should you pass this class? Y or N: ");
     while ((tmp = getchar()) != '\n')
               answer = tmp;

     switch(answer) {
     case 'Y':
     case 'y':
               printf("Good, you're on your way!\n");
               break;
     case 'N':
     case 'n':
               printf("... try again!\n");
               break;
     default:
               printf("That's not a Y or an N. ");
               printf("It's been one of those days, huh?\n");
     }

     return 0;
}
```

In the while loop you are assigning character input from stdin into `tmp`. If it is not a newline character, the character is assigned into `answer`. `answer` will contain the last character before the newline.

3 Lab Specifications

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, as the lab will be counted as late if you submit or resubmit it after the due date.**

3.1 Requirements

In order to complete Prelab 2, you will need the following:

- An understanding of how to compile programs (Lab 0)

- An understanding of how to read user input (Lab 1)

Complete the following before lab:

1. Read/Review Chapter 5 in *C Programming: A Modern Approach*.
2. Read Chapter 5 in The Linux Command Line.
3. Answer the questions in Section 3.2 of this prelab, and submit via Canvas by the due date.

3.2 Questions & Additional Problems

Exercise 1 (prelab2_debug.c).
Find as many errors as you can in the following code. Feel free to attempt compiling to help you locate the errors in this code. This method may also help you learn how the compiler responds to various programming errors. Type the original broken code and then add comments to show what corrections are needed. Save this file as `prelab2_debug.c`. Add a comment in your doxygen header comment at the top of the file with the number of errors you found.

```c
#include <stdio>

void main()
{
      char input;

      printf("Enter a character: \n");
      scanf("%f", input);

      switch (input) {
      case 'a':
              printf("A is for apple.\n");

      case 'b';
              printf("B is for Banana!/n");
              break;
      default:
              printf("The letter %d isn't important!\n", input);
      }

      return 0;
}
```

Exercise 2 (ex11.c, out-ex11.script, pp1.c out-pp1.script, pp7.c, out-pp7.script).
Complete the following problems taken from Chapter 5 of *C Programming: A Modern Approach*.

If the problem is an exercise, name the source code for these problems `exN.c`, where *N* is replaced by the exercise number.

If the problem is a programming project, name the source code for these problems `ppN.c`, where *N* is replaced by the programming project number.

1. **Exercise 11:** The following table shows telephone area codes in the state of Georgia along with the largest city in each area:

Area Code	*Major City*
229	Albany
404	Atlanta
470	Atlanta
478	Macon
678	Atlanta
706	Columbus
762	Columbus
770	Atlanta
912	Savannah

   Write a `switch` statement whose controlling expression is the variable `area_code`. If the `area_code` is in the table, the `switch` statement will print the corresponding city name. Otherwise the `switch` statement will display the message `Area code not recognized`. Make your `switch` statement as simple as possible. Name this file `ex11.c` and generate a script file named `out-ex11.script` showing the compilation and running of this program.

2. **Programming Project 1:** Write a program that calculates how many digits a number contains:

```
Enter a number: 374
The number 374 has 3 digits
```

   You may assume that the number has no more than four digits. *Hint*: Use `if` statements to test the number. For example, if the number is between 0 and 9, it has one digit. If the number is between 10 and 99, it has two digits. Name this file `pp1.c` and generate a script file named `out-pp1.script` showing the compilation and running of this program.

3. **Programming Project 7:** Write a program that finds the largest and smallest of four integers entered by the user:

```
Enter four integers: 21 43 10 35
Largest: 43
Smallest: 10
```

   Use as few `if` statements as possible. *Hint*: four `if` statements are sufficient. Name this file `pp7.c` and generate a script file named `out-pp7.script` showing the compilation and

running of this program.

You will submit these problems as part of the lab tarball.

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

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_groupNUMBER_prelab2.tar.gz`

Replace the `NUMBER` with your assigned group number and have one group member upload the tarball to Canvas before the due date.

Only one person in your group needs to upload your group's tarball to Canvas. If, upon being uploaded, all members of your group do not see that the file has been submitted, please contact the instructor or head TA as soon as possible.

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

List of Files to Submit