Prelab 2: An Introduction to Conditional Execution in C

CSE/IT 113

---

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 to 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 Requirements

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)

Complete the following before lab:

1. Read/Review Chapters 5 in *C Programming: A Modern Approach.*
2. Read Chapter 5 in The Linux Command Line.
3. Answer the questions in Section 4 of this prelab, and submit via Canvas by the due date.

3 Material

In order to complete this prelab, read the following material and answer the questions in Section 4.

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 met. In particular, you will be implementing an `if` statement and a `switch` statement, using conditions and logical expressions.

3.1 Conditions

Conditions are integral parts of programming, because they are the cornerstone to implementing flow control. A condition 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`, a false value is equal to `0`.

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

        return 0;
}
```

Remember:

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

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

3.1.2 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" evaluation of complex expressions to speed up program execution. If you had a complex **and** expression, once one of the conditions is false, there is no need to evaluate the rest of the expression as the entire expression is false. To save CPU cycles, C will not evaluate the rest of the conditions. This is known as "short-circuiting". Short-circuiting can lead to subtle runtime errors if you use assignment operations in your conditions.

3.2 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:

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

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.

3.2.1 Blocks

You have probably noticed that the **if** statement is framed by curly braces, **{** and **}**. 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 is executed.

3.2.2 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 a *bad coding style* as you are indenting needlessly. The condition involves a test of the same variable. In this case, it is much simpler to simplify the structure by using an **if/else if** structure:

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

A better use of nesting is once you have entered a body you have further processing to do on a *different* variable or variables. For example, this is a valid use of nesting:

```
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 */
        }
        /* todo */
}
else {
     /* none of the above is true; default case */
}
```

3.3 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 character or integer data-types you can use the **`switch`** statement. The following example is the previous **`if`**/**`else`** statement translated into a **`switch`** statement:

```
switch (foo) {
case 1: /* equivalent to test (foo == 1)  */
        /* todo -- executes if foo == 1*/
        break; /* execution moves to the } */
case 2:
        /* todo */
        break;
case 3:
        /* todo */
        break;
default:
        /* none of the above blocks executed
           the 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 and something different for even digits you could write code like this:

```
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 `char` or `int` data-types. That is, the data-type you are switching on `switch (data-type)` must be an `int` or a `char` data-type.

3.4 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 character data. Compile and run this simple program:

```
#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 string (technically an array of characters that are NULL terminated).

The following short program demonstrates `fgetc()` and a **switch** statement using character comparison:

```
#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("There's always a few in each class ... \n");
                break;
      default:
                printf("That's not a Y or an N. ");
                printf("It doesn't look good for you.\n");
      }

      return 0;
}
```

In the line 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.

4 Questions

Create and compile the following programs, create a script file for each program showing how you ran the program, and answer the questions at the end of this section.

Submit your source code, scripts and answers in a tarball file named

`cse113_firstname_lastname_prelab2.tar.gz`

to Canvas by the beginning of your lab section. Use the command

`tar czvf cse113_firstname_lastname_prelab2.tar.gz prelab2*`

to create the tarball. You can use the `t` option to view the files in the tarball

`tar tf cse113_firstname_lastname_prelab2.tar.gz`

Make sure to follow the Coding Style for the class. All examples conform to the Coding Style. Note where the braces are for **`if`** statements and **`switch`** statements and how the **`case`** statements line up underneath the **`switch`** statement.

1. Compile and run the following C program. You can use your own computer, a TCC computer, or a CS lab computer. For your script (`prelab2_q1.script`), you will need to run the program at least four times giving it the input `1` the first time, `2` the second time, `5` the third time, and any other number the fourth time. Modify this code to say something witty when the user enters a `5`, and also, add comments explaining what each block of code does. Save it as a file named `prelab2_q1.c` and create a matching script `prelab2_q1.script`.

```c
#include <stdio.h>

int main()
{
	int input = 0;

	printf("Enter an integer: \n");
	scanf("%d", &input);

	if (input == 1) {
		printf("First is the worst.\n");
	}
	else if (input == 2) {
		printf("Second is the best. ");
		printf("Because it's the first prime! ");
	}
	else {
		printf("I can't sing about %d.\n", input);
	}

	return 0;
}
```

2. Compile and run the following C code. For your script (`prelab2_q2.script`), you will need to run the program at least four times giving it the input `1` the first time, `3` the second time, `7` the third time, and any other number the fourth time. Modify this code to say something witty when the user enters a `7`, and also add comments explaining what each block of code does. Save it as a file named `prelab2_q2.c`.

```c
#include <stdio.h>

int main()
{
    int input = 0;

    printf("Enter an integer: \n");
    scanf("%d", &input);



    switch (input) {
    case 1:
        printf("One is the loneliest number.\n");
        break;
    case 3:
        printf("Three is just as bad as one. ");
        printf("But worse. Because it's 3!\n ");
        break;
    default:
        printf("I can't sing about %d.\n", input);
    }

    return 0;
}
```

3. Are the two programs above equivalent? Which program is easier for you to understand? Save your answers in a text file named `prelab2_q3.txt`.

4. Find as many errors as you can in the following code. Try finding them without compiling the code. Type the original broken code and then add comments to show what corrections are needed. Save this file as `prelab2_q4.c`. Add a 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;
}
```

5 Submission

Tar all your files, *.c , *.txt, *.script, into a single tarball named

`cse113_firstname_lastname_prelab2.tar.gz`

and upload to Canvas before the due date.