CpSc 1111 Lab 4
Formatting and Flow Control

Overview

By the end of the lab, you will be able to:

- use `fscanf()` to accept a character input from the user
- execute a basic block iteratively using loops to produce a multiplication table

Background Information

The following will review the three different types of loops that you learned about in lecture.

Loops

Remember lab 1 where you were supposed to print "Hello World!" 20 times? What if you were supposed to print that message 1000 times? Would you really copy and paste it 1000 times in your program? Sometimes it is necessary to repetitively execute a statement, or block of statements, while some condition remains true, or until some condition becomes false.

Because of all the possible formulations of conditions, it is the case that there are multiple correct ways to construct such iterations. And, alas, there are even more incorrect ways to construct them.

There are three different iteration constructs in C:

1. while loop
2. do-while loop
3. for loop

---

1. `while` Loop

The `while` loop allows us to write a segment of code that repeats *while* some condition remains `true`.

Structure:

```
// loop index variable used in condition declared and
// initialized somewhere above the while loop
while (condition)
{
   one-or-more statements;

   loop_expression;  // expression that changes the condition,
                     // updates the loop index variable
}
```

Important Notes:

- The statement or statements controlled by the `while` loop are called the body of the loop and are enclosed by curly braces.
- It is necessary that the body of the loop modify the value of the condition. Why?
- Proper use of indentation is critical for human readability.
- But, indentation is completely irrelevant to the C compiler.
- Example: Print the integer values between 10 and 1 in decreasing order. (**Note:** This is not a complete C program; it needs to include `stdio.h` and it needs a proper `main()` function header. Also, it is not indented properly – see how hard it is to read and make sense of the code?)

```
int value;
value = 10;
fprintf(stdout, " Table Of Values \n");
while (value > 0) {
fprintf (stdout, "%3d\n", value);
value = value - 1;
}
```

- In the above code, we only want the heading to be printed one time, so the first `fprintf()` is placed outside the loop.

Example of `while` loop (with proper formatting and good use of comments):

```
// addition.c
// 08/31/15
// Print the sum of positive integers provided by the user.

#include <stdio.h>

int main(void) {
    int input;   // to hold user input
    int sum = 0; // sum of all input values

    // prompt user and get input
    printf("Enter a positive integer, or a negative integer when done: ");
    scanf("%d", &input);

    // loop to add each input to sum, and get another integer from user
    while (input >= 0) {
        sum = sum + input;
        printf("Enter a positive integer, or a negative integer when done: ");
        scanf("%d", &input);
    }

    // display the sum
    printf("The sum is %d\n", sum);

    return 0;
}
```

Notice that we prompt the user for the first input before the loop. Inside the loop, we do the math and prompt the user for the next number.

---

2. `do-while` Loop

The `do-while` loop allows us to write a segment of code that also repeats *while* some condition remains true.

Structure:

```
// loop index variable used in condition declared and
// initialized somewhere above do-while loop
do
{
   one-or-more statements;

   loop_expression;  // expression that changes the condition, or updates
                     // the loop index variable
} while (condition);
```

Important Notes:

- The above loop is guaranteed to execute the statement(s) at least once. Do you see why?
- As with the `while()` loop, it is necessary that the body of the loop modify the value of the condition so that it is not an infinite loop.
- Don't forget the semi-colon after the while condition. The other loops do not have a semi-colon at the end of the loop.

---

3. `for` Loop

The third loop structure is the `for` loop.

Structure:

```
// loop index variable declared somewhere above for loop
for (init_expression; loop_condition; loop_expression)
{
   one-or-more statements;

}
```

Important Notes:

- The statement or statements controlled by the `for` loop are called the body of the loop and are enclosed in curly braces.
- The **`init_expression`** gives the starting value for the control variable (loop index variable) being used by the loop. The control variable needs to be declared somewhere above the `for` loop, probably at the top of the program where the other variables are declared.
  Example:
  ```
  int i;
  for (i = 0; loop_condition; loop_expression)
  ```
- The **`loop_condition`** will be evaluated before each iteration of the loop; if it evaluates to true, an iteration of the loop body will occur; if it evaluates to false, no code inside the body of the loop will execute.
  Example:
  ```
  int i;
  for (i = 0; i <= 3; loop_expression)
  ```
  - In this code above, as long as the value of `i` is less than or equal to 3, the statements inside the body of the loop will be executed.

- The **loop_expression** will change the value of the loop control variable so that a stopping point may be reached in the loop_condition.
  Example:
  ```
  int i;
  for (i = 0; i <= 3; i++)
  ```
  - In the code above, i will increment by 1 each time immediately after an iteration of the loop, allowing for the loop_condition to eventually evaluate to false (assuming a proper condition exists), thereby ending the loop. (i++ is the same as i = i + 1 and the same as i += 1 and with loops, the same as ++i)

Example:
```
for (value = 10; value > 0; value--) {
    fprintf(stdout, "%3d\n", value);
}
```

Note: value-- is the same as value = value - 1
Also, the starting value for the control variable called value in this case is 10, and it decrements by 1 after each iteration of the loop. The loop will continue to iterate as long as the value of the control variable is greater than 0. So, how many iterations will occur?

Review of scanf()

By now, you know to use scanf() or fscanf() to get input from the user. For example, if you were asking the user to enter an integer, and storing their input into a variable called input, your code might look like the following:

```
int input;
fprintf(stdout, "Enter an integer:  ");
fscanf(stdin, "%d", &input);
```

If the user were to enter a character, your code would change this way:

```
char inputChar;
fprintf(stdout, "Enter a character:  ");
fscanf(stdin, "%c", &inputChar);
```

Lab Assignment

Reminder About Style, Formatting, and Commenting Requirements

- The top of your file should have a header comment, which should contain:
  - Your name
  - Date
  - Lab section
  - Lab number
  - Brief description about what the program does
  - Any other helpful information that you think would be good to have.
- Variables should be declared at the top of the main function, and should have meaningful names.
- One exception to the rule of using meaningful names for variables is to use i or j or n or some other single letter as a loop index variable name; this is a common, acceptable practice.
- Always indent your code in a readable way. Some formatting examples may be found here: https://people.cs.clemson.edu/~chochri/Assignments/Formatting_Examples.pdf
- Don't forget to use the -Wall flag when compiling, for example: gcc-6 -Wall lab4.c

Create a file called `lab4.c`. In it, provide a C program that will print a multiplication table to the screen. You will first prompt the user to enter an integer, which will be used as the size of the multiplication table. For example, if the user enters a 5, it will show the product of two values ranging from 1 to 5, which should look something like the following:

```
Enter an integer to display the multiplication table from 1 to N: 5

   X   1   2   3   4   5
   1   1   2   3   4   5
   2   2   4   6   8  10
   3   3   6   9  12  15
   4   4   8  12  16  20
   5   5  10  15  20  25
```

If the user enters an 8, it will be from 1 to 8, which should look something like the following:

```
Enter an integer to display the multiplication table from 1 to N: 8

   X   1   2   3   4   5   6   7   8
   1   1   2   3   4   5   6   7   8
   2   2   4   6   8  10  12  14  16
   3   3   6   9  12  15  18  21  24
   4   4   8  12  16  20  24  28  32
   5   5  10  15  20  25  30  35  40
   6   6  12  18  24  30  36  42  48
   7   7  14  21  28  35  42  49  56
   8   8  16  24  32  40  48  56  64
```

Keep in mind for the numbers to align in each column, each number must be evenly spaced regardless of the length of the number. In the above examples the integers displayed are evenly spaced 4 characters apart, this can be done by using the `%4d` specifier with `printf()` or `fprintf()`.

Turn In Work

1. Before turning in your assignment, make sure you have followed all of the instructions stated in this assignment and any additional instructions given by your lab instructor(s). Always test, test, and retest that your program compiles and runs successfully on our Unix machines before submitting it.
2. Show your TA that you completed the assignment. Then submit your `lab4.c` program using the handin page: http://handin.cs.clemson.edu. ***Don't forget to always check on the handin page that your submission worked. You can go to your bucket to see what is there.***

Grading Rubric

If your program does not compile on our Unix machines or your assignment was not submitted on time, then you will receive a grade of zero for this assignment. Otherwise, points for this lab assignment will be earned based on the following criteria:

Functionality (correct output)	80
Style, Formatting, & Commenting	10
Use of loops for table	10

Possible loss of points for other things, such as warnings when compile (-5), or other penalties for not following this lab's instructions.