Tracing and Visualizing C Programs Using C Coding Tutor: Learning Activities for CCPROG1

Preliminary Task: please click this link for Check-up Quiz #6 and answer a non-graded check-up quiz on nested loops. The form is only accessible to a DLSU domain; please log-in to your DLSU account in case you cannot access it the first time around. The quiz assumes that you have already covered the topic on nested loops in your CCPROG1 class. The objective of the quiz is to determine possible misconceptions which we intend to clarify and remediate as part of the learning activities. The answers to Quiz #6 are given on page 4 of this document.

MODULE 6: Tracing and Visualizing Nested Loops

A piece of code where the body of a loop contains another loop construct is referred to as a nested loop. A loop within a loop is also sometimes called a double loop, and a loop within a loop within a loop is also referred to as triple loop. Try to become proficient in developing algorithms and programs with nested loops because they are going to be heavily used in your next level subjects such as CCPROG2 (advanced programming) and CCDSALG.

Learning Activity 5.1: Printing a Grid of Asterisks

Let's consider the task of writing a program that will produce the following as output, i.e., 3 rows and 5 columns of asterisks (or any other character that you may choose).

```
*****
*****
*****
```

We'll present several solutions starting from naïve/brute force solution to the final generalized solution that uses a nested loop.

Learning Outcome: The student can trace a program with a nested loop

Instructions:

1. A naïve or brute force solution need not use any loop at all, and use just one **printf** statement as shown in Figure 6.1 But let's expose the development of our eventual final solution on a program that uses 3 **printf** statements, with one **printf** for each row of asterisks as shown in the Figure 6.2. Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor to see the result in the Print Output window.

```c
#include <stdio.h>

int main()
{
    printf("*****\n*****\n*****\n");
    return 0;
}
```

Figure 6.1: Solution #1: naïve or brute force solution with just one printf statement

```c
#include <stdio.h>

int main()
{
    printf("*****\n");
    printf("*****\n");
    printf("*****\n");
    return 0;
}
```

Figure 6.2: Solution #2: still a naïve or brute force solution with 3 printf statements

2. The naïve solution is already a correct solution. However, it is static, i.e., fixed to an output of 3 rows and 5 columns of asterisks. What if we want to increase or decrease the number of rows and/or columns? If we, say, want to output 1000 rows of asterisks, then a brute solution will require adding 997 more **printf** statements following the logic above.

   Notice that there is a repetitive pattern where **printf** outputs the same string of asterisks three times. Remember that codes that are repetitive are, in general, better re-implemented with a loop as shown in Figure 6.3. Click this link to open a browser window with a pre-encoded program that outputs the required result using a single **for** loop. Run/trace the program in C Tutor to see the results in the Print Output window.

   The disadvantage of the single **for** loop solution is that it requires more steps, specifically for testing the condition if it is true or false, and for incrementing the counter. So, what's "good" about this solution? Well, it generalizes better, i.e., it will not take a lot of coding effort to increase or decrease the number of rows of asterisks. For example, if want to output 1000 rows of asterisks, all we need to do is to change one line of code, specifically, the condition from **i < 3** to **i < 1000**. Do try this in DEV-CPP or VSC IDE (not in C Tutor because tracing will take a lot of steps).

```c
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 3; i++)
        printf("*****\n"); // repeat the body of the loop 3 times

    return 0;
}
```

Figure 6.3: Solution #3: use a single for loop to generate 3 rows of asterisks

3. Let's now look at a single row of asterisks. Notice that each row of output is also repetitive! Instead of a single **printf** statement for printing 5 asterisks, we can re-implement it with another **for** loop where the body of the loop is repeated 5 times. The body of the loop in this case is a **printf** statement that prints only one asterisk. At the end of the loop, there's another **printf** that outputs the newline character '\n'. Figure 6.4 shows a screenshot of a code with **PrintOneLine** function achieving the idea that we described.

   Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor. Pay special attention to the values of the counters **i** and **j** as you progress through each step.

```
1   #include <stdio.h>
2
3   // print one line of 5 asterisks
4   void PrintOneLine(void)
5   {
6       int j;
7       for (j = 0; j < 5; j++)
8           printf("*"); // repeat the body of this loop 5 times
9
10      printf("\n");
11  }
12
13  int main()
14  {
15      int i;
16
17      for (i = 0; i < 3; i++)
18          PrintOneLine(); // repeat the body of this loop 3 times
19
20      return 0;
21  }
```

Figure 6.4: Solution #3: use a single for loop in **PrintOneLine** function to generate one line of 5 asterisks

4. It may not be obvious, but the latest code has a nested loop, i.e., there is a **for** loop inside the body of another **for** loop. To see this, let's remove the function call to **PrintOneLine**, and replace it instead with the body of **PrintOneLine**. We will also remove the **PrintOneLine** function definition. Figure 6.5 shows visually that the inner loop bounded by the yellow rectangle is inside the outer loop bounded by the red rectangle. There is only one statement in the body of the inner loop, specifically the **printf** statement in Line 11. There are two statements inside the body of the outer loop, specifically the inner for loop, and the **printf** statement in Line 13.

```c
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 3; i++)  {  // this is the outer loop
        int j;

        for (j = 0; j < 5; j++) // this is the inner loop
            printf("*");

        printf("\n");
    }

    return 0;
}
```

Figure 6.5: Solution #4: nested loop solution

Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor. Again, pay attention to the values of the counters **i** and **j** as you progress through each step. Variable **i** is the counter for the outer loop, and variable **j** is the counter for the inner loop.

Please answer the following questions based on the given code (see Figure 6.5). Note that these are the same questions asked in Check-up Quiz #6. The answers can be found in the footnote on this page[1].

Q1: How many times will the initialization **i = 0** be performed?
Q2: How many times will the condition **i < 3** be tested?
Q3: How many times will the condition **i < 3** be evaluated as true?
Q4: How many times will the condition **i < 3** be evaluated as false?
Q5: How many times will the increment operation **i++** be performed?
Q6: How many times will the initialization **j = 0** be performed?
Q7: How many times will the condition **j < 5** be tested?
Q8: How many times will the condition **j < 5** be evaluated as true?
Q9: How many times will the condition **j < 5** be evaluated as false?
Q10: How many times will the increment operation **j++** be performed?
Q11: How many times will the **printf("*")** be performed?
Q12: How many times will the **printf("\n")** be performed?

[1] A1: 1
A2: 4 (tested as 0 < 3, 1 < 3, 2 < 3, 3 < 3)
A3: 3 (when 0 < 3, 1 < 3 and 2 < 3)
A4: 1 (when 3 < 3)
A5: 3 (incremented from 0 to 1, from 1 to 2, and finally from 2 to 3)
A6: 3 (when i is 0, when i is 1 and when i is 2)
A7: 18 (6 times when i is 0, 6 times when i is 1 and 6 times when i is 2)
A8: 15 (5 times when i is 0, 5 times when i is 1 and 5 times when i is 2)
A9: 3 (1 time when i is 0, 1 time when i is 1 and 1 time when i is 2)
A10: 15 (5 times when i is 0, 5 times when i is 1 and 5 times when i is 2)
A11: 15
A12: 3

5. The current implementation is fixed to 3 x 5 grid of asterisks. What if we want to change both the number of rows and columns, and even the character used for printing? We can achieve that by defining a function with three parameters, specifically, the number of rows, the number of columns and the character symbol as shown in Figure 6.6.

   Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor.

```c
#include <stdio.h>

void PrintGrid(int nrows, int ncols, char ch)
{
    int i;

    for (i = 0; i < nrows; i++)  {  // this is the outer loop
        int j;

        for (j = 0; j < ncols; j++) // this is the inner loop
            printf("%c", ch);

        printf("\n");
    }
}

int main()
{
    PrintGrid(3, 5, '*');  // print a 3x5 grid of *
    printf("\n");
    PrintGrid(2, 4, '#');  // print a 2x4 grid of #

    return 0;
}
```

Figure 6.6: Solution #5: more generalized solution with parameters for the number of rows, number of columns and the character to be printed

6. Experiment by modifying the function call parameters and see the effects on the output.

Learning Activity 5.2: Printing a Right Triangle Shape

Let's next consider the task of writing a program that will produce a triangle shape as output, as shown for example below.

```
*
**
***
****
*****
```

Can you identify the pattern?

The pattern here is that the 1st line contains 1 asterisk, the 2nd line contains 2 asterisks, the 3rd line contains 3 asterisks, and so on. In general, the **i**'th line contains **i** number of asterisks.

Just like in the previous activity, we'll present several solutions starting from naïve/brute force solution to the final generalized solutions that uses a nested loop.

Learning Outcome: The student can trace a program with a nested loop

Instructions:

1. You already have an idea of the naïve or brute force solution, right? You probably have thought of a program like the code in Figure 6.7.

   Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor to see the result in the Print Output window.

```c
#include <stdio.h>

int main()
{
    printf("*\n");     // 1st line has 1 asterisk
    printf("**\n");    // 2nd line has 2 asterisks
    printf("***\n");   // 3rd line has 3 asterisks
    printf("****\n");  // 4th line has 4 asterisks
    printf("*****\n"); // 5th line has 5 asterisks
    return 0;
}
```

Figure 6.7: Solution #1: a naïve or brute force solution with 5 printf statements

2. An improvement is to use the same idea presented in **PrintOneLine** function in the previous activity. The modification is that we pass an integer parameter that represents the number of characters to be printed in one line. We then call **PrintOneLine** five times in the **main** function. Refer to Figure 6.8 for the codes.

   Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor to see the result in the Print Output window. Pay attention to the values of parameter **n** and counter **j** as you progress through each step.

```c
#include <stdio.h>

// print one line of n asterisks
void PrintOneLine(int n)
{
    int j;
    for (j = 0; j < n; j++)
        printf("*"); // repeat the body n times

    printf("\n");
}

int main()
{
    PrintOneLine(1);
    PrintOneLine(2);
    PrintOneLine(3);
    PrintOneLine(4);
    PrintOneLine(5);

    return 0;
}
```

Figure 6.8: Solution #2: use a for loop to generate n number of asterisks per line

3. Notice that the actual parameters values to the **PrintOneLine** function calls in Lines 15 to 19 are 1, 2, 3, 4 and 5 which means that we can make use of a for loop as well to generate this number sequence. Figure 6.9 shows the modified code.

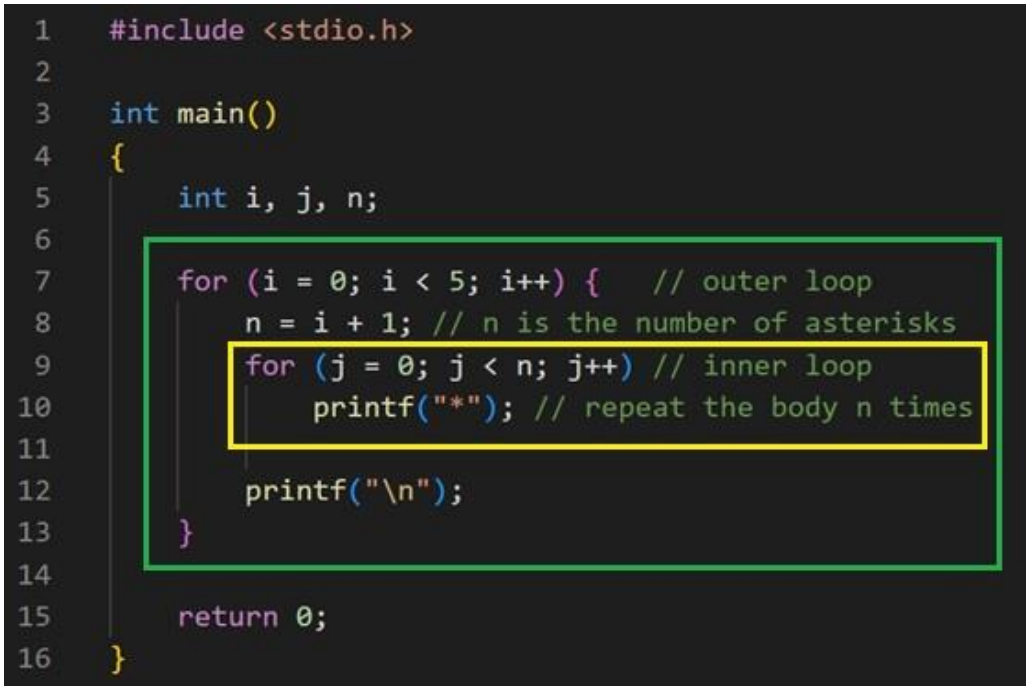
   Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor to see the result in the Print Output window. Just like before, pay attention to the values of parameter **n** and counters **i** and **j** as you progress through each step.

```
1   #include <stdio.h>
2
3   // print one line of n asterisks
4   void PrintOneLine(int n)
5   {
6       int j;
7       for (j = 0; j < n; j++)
8           printf("*"); // repeat the body n times
9
10      printf("\n");
11  }
12
13  int main()
14  {
15      int i;
16
17      for (i = 0; i < 5; i++)
18          PrintOneLine(i + 1);
19
20      return 0;
21  }
```

Figure 6.9: Solution #3: use a for loop to generate the numbers 1 to 5 in Line 18

4. Just like in the previous activity, we can rewrite the program to explicitly show a nested loop solution by replacing the function call to **PrintOneLine** by the body of **PrintOneLine** as shown in Figure 6.10. The outer for loop is shown visually bounded by the green rectangle, and the inner loop is shown bounded by the yellow rectangle.

   Click this link to open a browser window with a pre-encoded program and run/trace the program in C Tutor to see the result in the Print Output window. Just like before, pay attention to the values of variable **n** and counters **i** and **j** as you progress through each step.



```
1   #include <stdio.h>
2
3   int main()
4   {
5       int i, j, n;
6
7       for (i = 0; i < 5; i++) {   // outer loop
8           n = i + 1; // n is the number of asterisks
9           for (j = 0; j < n; j++) // inner loop
10              printf("*"); // repeat the body n times
11
12          printf("\n");
13      }
14
15      return 0;
16  }
```

Figure 6.10: Solution #4: a loop within a loop is shown explicitly

5. Challenge: now it's your turn to implement a more generalized solution. Instead of fixing the triangle height to 5 lines, make it generalized. Implement a function **PrintTriangle(height)** where the height is the triangle height (we assume it to be a positive integer).

   Click this link and fill in the missing codes. Run/test and trace your program in C Tutor.

FAREWELL NOTES/HINTS:

- The technique, as you should have learned by now, is to **find the pattern**.
- Instead of writing a nested loop solution right away, it is suggested that you encapsulate the logic of the inner loop as a function first. This can help simplify the thinking process.
- It is common among many C programmers to use counter variables named as **i**, **j** and **k** in nested loops. Of course, you may use other names that you like.
- Be careful when updating the counter variables! Novice programmers often make the mistake of modifying variable **i** instead of **j** as shown in the code below:

```
for (i = 0; i < 3; i++) {
    for (j = 0; j < 5; i++)  // Logical error -- should increment j not i.
            :
}
```

- All the **for** loop codes above can be replaced by a **while** loop or a **do while** loop.
- It is suggested that you complete the exercises given in your CCPROG1 Canvas online notes. Focused practice is essential in mastering nested loops and all the topics covered in the previous modules.
- If you encounter a difficult-to-debug code, it may be helpful to visualize and trace it using C Tutor…

--- End of Module 6 (this is the last module) ---