Achieve Target 1

We have seen different kinds of loop structures. Let’s see more variations on loop structures.

```c
#include <stdio.h>

void main() {
  int col;
  for (col = 0; col < 6; col++) {
    printf("**");
  }
}
```

**Output**

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

6 asterisks

The above is a simple for loop. Let’s work out how many times is the for loop body (the blue statement) executed? By counting we have got 6. The programmer’s way is to carefully look at the for statement. The variable `col` starts from 0, the loop stops when `col` is 6. How many loops would be executed before `col` becomes 6? Use your fingers and count it!

Modify the program so that it prints 10 asterisks instead. **Work** on the program to make it happen.

Remember this pattern for writing counter-controlled loops. The initialisation of the counter to 0 and the use of less-than operator (<) in the looping condition is a good way to follow. Because the number following the less-than operator would be a good indication of the number of loops. But **be careful**, the loop counter must be increased one per loop. **Remember** it.
Achieve Target 1 Cont

We can print patterns by crafting the for loop body more carefully.

```c
#include <stdio.h>

void main() {
    int col;
    for (col = 0; col < 6; col++) {
        if (col % 2)
            printf("#");
        else
            printf("*");
    }
}
```

The above pattern shows that an asterisk is printed when the counter (variable `col`) is even, and a hash (`#`) is printed when the counter is odd. The statement in blue checks whether `col` is an odd number. The expression

```
col % 2
```

is equivalent to, in a if statement,

```
col % 2 != 0
```

The if statement only cares whether the value is zero or non-zero. No benefit is added by checking if a value is not zero, because a non-zero would result in one (which is non-zero) and zero would result in zero. Remember it.

Modify the program so that it prints the following pattern. Work on the program to make it happen.

```
###*###*###
```
Achieve Target 2

We have put lots of different things inside a loop body. So there is no reason why we cannot put another loop inside a loop body. The following program shows that one for loop is placed inside another for loop.

```
#include <stdio.h>

void main() {
    int row, col;
    for (row = 0; row < 8; row++) {
        for (col = 0; col < 6; col++) {
            printf("*");
        }
    }
}
```

Output

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

How many times is the inner for loop body (the blue statement) executed? We could simply count the number of asterisks printed! There are 48 asterisks. Rather than counting the asterisks, we could look at the loops carefully. The inner for loop runs the blue statement 6 times when executed. The outer for loop runs the inner for loop 8 times. By simple mathematics, we have 8 times 6 that is 48.

We could instead have one for loop that executes 48 loops, but having a loop within another loop can generate complex patterns.

```
#include <stdio.h>

void main() {
    int row, col;
    for (row = 0; row < 3; row++) {
        for (col = 0; col < 5; col++) {
            printf("*");
        }
        printf("A");
    }
}
```

```
*****A*****A*****A
```

There are 3 times 5 numbers of asterisks. After every 5 asterisks, an ‘A’ is printed by the bold blue statement. The outer for loop executes 3 times, so there are 3 ‘A’s in total.

Modify the program to produce the following two patterns. **Work** on the program to make it happen.

```
B*****AB*****AB*****A
B*##*##*##AB*##*##*##AB*##*##*##*##AC
```
Achieve Target 2 Cont

The following program prints a two-dimensional pattern.

```c
#include <stdio.h>

void main() {
    int row, col;
    for (row = 0; row < 3; row++) {
        for (col = 0; col < 5; col++) {
            printf("*");
        }
        printf("\n");
    }
}
```

The 2D pattern is printed with suitable addition of newline characters (\n) to split the asterisks into multiple rows. The program could be rewritten using while loops, but for loops are clearer to read and write.

Modify the program so that two while loops are used. Work on the program to make it happen.

The size of the square and the character to print can both be modified. Modify the above program (for loops) to print a square of plus sign '+' characters with 4 rows and 3 columns. Work on the program to make it happen.
Achieve Target 3

We can generate more complex 2D patterns with two or more loops. The following program is an example.

```c
#include <stdio.h>

void main() {
    int row, col;
    for (row = 0; row < 4; row++) {
        for (col = 0; col < 10; col++) {
            if (col % 2)
                printf("O");
            else
                printf(" ");
        }
        printf("\n");
    }
}
```

The program made use of the even-ness of the column counter (variable `col`) to print different character. When `col` is even, a space is printed, and when `col` is odd, an 'O' is printed.

Modify the program so that it prints the following pattern instead. There are 6 rows and 5 columns. There is one empty row between two rows of 'O'. Work on the program to make it happen.

```
OOOOO
OOOOO
OOOOO
```

The following pattern is a combination of the above two. It has 8 rows and 8 columns. Modify the program again so that it prints the following pattern. Work on the program to make it happen.

```
* * * *
* * * *
* * * *
* * * *
* * * *
* * * *
* * * *
* * * *
```
Achieve Target 3 Cont

The following program has three level of for loops. It looks complex but you could try identifying different loop structures and their function.

```c
void main() {
    int row, col;
    int repeat;
    for (row = 0; row < 8; row++) {
        for (repeat = 0; repeat < 4; repeat++) {
            if (repeat % 2 == 0) {
                for (col = 0; col < 8; col++)  {
                    if (((row + col) % 2)
                        printf("*");
                    else
                        printf(" ");
                } /* for col */
            } else {
                for (col = 0; col < 8; col++) {
                    printf(" ");
                } /* for col */
            } /* if-else */
        } /* for repeat */
    } /* for row */
    printf("\n");
}
```

Putting one loop inside another loop is usually referred as nested loops. Nested loops are very useful for complex data processing (in addition to printing patterns). These patterns however allow you to inspect how nested loops work.

The loop of counter `repeat` allows the repeated printing of the patterns across the rows. The value of variable `repeat` determines one of the two patterns to print. If `repeat` is an even number, then the checkerboard pattern is printed; otherwise only spaces are printed.

Modify the program so that it prints the following pattern. Work on the program to make it happen.
Achieve Target 4

Let's leave loops for a moment and work on a new thing.

The following program allows us to see some interesting effects of mixing the AND or OR operators (`&&`, `||`) with post-increment or pre-increment operators (`, `).

```c
#include <stdio.h>

void main() {
    int a = 25;
    int b = 10;
    if ((a++ > 25) && (b++ > 20)) {
        printf("OK\n");
    }
    printf("a = %d b = %d\n", a, b);
    getchar();
}
```

**Work** out the values of `a` and `b` at the `printf` statement.

Check your answer with the solution document now. Most of you would get the *not-so-right* answer. It is not fair to say that you are wrong because you have not learned a special rule in the execution of the operators. The `b++` operation is not executed at all because of the `&&` operator.

Remember that the `&&` operator evaluates to 0 if any of the values is 0 (study the previous Reform Exercise if you do not remember). The first value `(a++ > 25)` is evaluated to 0 (because `a` is 25), so the outcome of the `&&` operator is fixed to 0 regardless of the second value `(b++ > 20)`. There is no need to evaluate the second value and therefore not executed. As a result, the `b++` operation is not executed. The `||` operator has this effect too. If the first value is 1, then the result of the `||` operator must be 1. There is no need to evaluate the second value. **Remember** it.

```c
a = 25; b = 10;
(a++ > 20) && (b++ > 20);
printf("a = %d b = %d\n", a, b);

a = 25; b = 10;
(a++ > 25) || (b++ > 20);
printf("a = %d b = %d\n", a, b);

a = 25; b = 10;
(a++ > 20) || (b++ > 20);
printf("a = %d b = %d\n", a, b);
...
```

**Work** out what are the values of `a` and `b` printed at the `printf` statements, taken into considerations of the special effect of the `&&` and `||` operators.

**Discuss** the advantages and disadvantages of this special effect

It is risky to mix the the AND or OR operators (`&&`, `||`) with post-increment or pre-increment operators (`, `) in one statement. It is a good practice to avoid it. **Remember** it.
Achieve Target 5

Let's come back to loops. The following program illustrates more on using loops.

```c
...  
void main() {  
    int count;  
    float sum; /* we use float here because assignment scores have 0.5 */  
    float score;  
    float highest;  

    count = 1;  
    sum = 0;  
    highest = 0;  

    while (count <= 4) {  
        printf("Enter score for assignment %d: ", count);  
        scanf("%f", &score);  
        sum = sum + score;  

        if (score > highest)  
            highest = score;  
        count = count + 1;  
    }  

    printf("Average score is %f and highest is %f\n", sum/4, highest);  
    getchar();  
...  
```

The above program reads in four assignment scores and reports the average score and the highest score. The detection of highest number is the new thing that is actually a common problem in programming.

The counter variable count controls the loop to repeat four times. The variable highest keep the highest score found so far. In each iteration, it is compared to the scores entered and is always assigned the larger value. The variable highest is initialised to 0 and this is significant. The value zero is the lowest possible score and any score entered should be greater or equal to it. This allows the variable highest to use as the base for comparison. Learn it.

Modify the program so that it reports the lowest score as well. You are reminded to take care of the following things: declaring a variable, give the variable a suitable initialisation, and add logic code so that the variable helps detecting the lowest score. Work on the program to make it happen.
Achieve Target 5 Cont

The following program is the HSI average calculator discussed in earlier Perform Session.

```c
#include <stdio.h>

void main() {
    float data = 0;
    float sum = 0;
    int count = 0;
    int holidaycount = 0;

    printf("Closing HSI Average for One Week (Enter -1 if holiday)\n");
    while (count < 5) {
        switch (count) {
            case 0: printf("Enter Monday HSI: "); break;
            case 1: printf("Enter Tuesday HSI: "); break;
            case 2: printf("Enter Wednesday HSI: "); break;
            case 3: printf("Enter Thursday HSI: "); break;
            case 4: printf("Enter Friday HSI: "); break;
        }
        scanf("%f", &data);
        if (data == -1) {
            holidaycount++;
            count++;
            continue;
        }
        sum = sum + data;
        count++;
    }
    if (count - holidaycount > 0)
        printf("Average is %f\n", sum/(count - holidaycount));
    getchar();
}
```

Modify the program so that it also reports the highest and the lowest HSI for the week. Be careful with the location in the while loop where you perform the comparison. **Work** on the program to make it happen.
Achieve Target 6

The following program shows more on handling input with a loop structure.

```c
#include <stdio.h>

void main() {
    int ch;
    int count = 0;
    printf("I will say what you say\n");
    ch = getchar();
    while (ch != '\n') {
        printf("%c", ch);
        count++;
        ch = getchar();
    }
    printf("\n");
    printf("Characters entered = %d\n", count);
    fflush(stdin);
    getchar();
}
```

I will say what you say
abcdef
abcdef
Characters entered = 6

The program will read one line of input and echo. So effectively the characters entered are printed in the following line. The loop also counts how many characters presents in the line, and prints it out at the end.

The expressions in blue are something new to you. You have seen `getchar` before for pausing the program and waiting for user to press a key. In fact, `getchar` is a function that returns one character entered from the keyboard. In the program, the character is stored in the variable `ch`.

Let’s look at the looping condition closely. The looping condition says that it continues to loop if the variable `ch` is not equal to the newline (`\n`) character. In other words, it continues to loop if end of the line is not seen yet. Each loop reads in one character and then prints it out. The variable count is then used to keep track of how many characters are read in so far.

We mentioned in another place that for input-controlled loops we need two statements for input. One is just before the loop and another at the end of the loop body. This program follows the pattern.

Modify the program so that it counts the total number of ‘a’ and ‘A’ found in the input. You will need an `if` statement. Work on the program to make it happen.
Achieve Target 6 Cont

The following program is functionally the same as the previous program.

```c
#include <stdio.h>

void main() {
    int ch;
    int count = 0;
    printf("I will say what you say\n");

    while ((ch = getchar()) != '\n') {
        printf("%c", ch);
        count++;
    }
    printf("\n");

    printf("Characters entered = %d\n", count);
    fflush(stdin);
    getchar();
}
```

There is now only one `getchar` instead of two. So this breaks the pattern of input-controlled loops.

The part in blue calls the function `getchar` and returns one character in the input. The character is assigned to the variable `ch`. Note that the expression in blue also produces a value (see the earlier Perform Session 2 for revision). The value is one produced by the assignment operator (=). The value is the one involved in assignment which is the value returned by `getchar`. Effectively the looping condition checks if the character entered is a newline character.

This is a convenient way to write input-controlled loops. It is worth to memorize how to write this loop because it is a very common pattern in file input/output programs. Note that we cannot do likewise with `scanf`. The function `scanf` does not return the input value as `getchar` does. So we cannot build an expression for the looping condition with `scanf`. Remember it.

Modify the program so that it echoes only the first, third, fifth characters in the entered line. In other words, it will prints out the first character for each pair of input characters. You should make use of the variable `count`. Work on the program to make it happen.
Achieve Target 7

In the last 6 targets, you have worked hard on the following ideas. Ponder upon these ideas and remember what you have learned.

1. It is very useful to learn how to calculate the number of loops that will be executed for a given loop structure. We could predict the number of loops in counter-controlled loops. However, the number of loops cannot be determined for input-controlled loops. The actual number of loops depends on the input length.

2. Nested loops could be used to do complex processing and print interesting patterns. Sometimes you feel that you need a nested loop but may find difficult in writing one. First, decide whether you need a nested loop first. You could do it by looking at the problem at hand and work out the patterns. The number of patterns present decides how many loops you need. It is better to illustrate it with examples.

The following have one pattern (or first-order pattern).

```
************
1234567890
aaaaaaaaaaaa
```

The following have two patterns working together (or second-order pattern)

```
*****     *****     *****
12341234123412341234
aaaabbbcccccdddddeeee
12342345345645675678
```

If there is only one pattern at work, only one loop is required. For two patterns at work together, then you need a two-level nested loop.

3. On expressions concerning AND (&&) and OR (||) operators. The skipping of the execution of the second operand (and the other operands) because the result of the operator is decided already with the first operand is known as short-circuit evaluation. This is one way to improve the speed of execution of program by not executing something no longer needed to know. The drawback is the potential problem with pre-increment or post-increment operators found in the second and the remaining operands. Avoid using pre-increment or post-increment operators with AND and OR operators.

4. When detecting the lowest or highest values inside a loop structure, you will always need variables to temporarily store the highest and lowest values found so far. The variables must be initialised properly. The one to store the lowest value must be initialised to the highest possible value in C. You could use FLT_MAX in the float.h library for floating point values. The one to store the highest value must be initialised to the smallest possible value in C. You could use -FLT_MAX for the value.

5. The function getchar returns one character found in the input. Note that it will not see the characters entered on the keyboard immediately. It will only see the characters when the Enter key is pressed on the keyboard. When users press keys on the keyboard, the characters are stored in a buffer. Only when the Enter key is pressed, then the characters in the buffer are passed on to the program.