We begin this session with something new and important than what we could do before.

```c
#include <stdio.h>
void main() {
    if (0)
        printf("One\n");
    if (1)
        printf("Two\n");
    while (0)
        printf("Three\n");
    while (1)
        printf("Four\n");
    getchar();
}
```

It is clear that the **statements in red** are not executed. Use your knowledge learned about **if statement** now.
You should see that the value 0 in the if statement disallows the execution of the first printf statement.

```c
#include <stdio.h>
void main() {
    if (0)
        printf("One\n");
    if (1)
        printf("Two\n");
    while (0)
        printf("Three\n");
    while (1)
        printf("Four\n");
    getchar();
}
```

The value 1 in the second if statement allows the execution of the second printf statement.

You could see from the output that the while statement operates in a similar way.

The value 0 in the while statement also disallows the execution of the following printf statement.

The value 1, following the same pattern, allows the execution to happen.

The significant difference is that the execution is repeated.

So you could see "Four" is printed repeatedly.
Achieve Target 1

We now want the program to give the following output, with "Three" printed repeatedly.

```c
#include <stdio.h>

void main() {
    if (0)
        printf("One\n");

    if (1)
        printf("Two\n");

    while (0)
        printf("Three\n");

    while (1)
        printf("Four\n");

    getchar();
}
```

Work on the program to make it happen.
Achieve Target 1

Solution

```c
#include <stdio.h>

void main() {
    if (1)
        printf("One\n");
    if (1)
        printf("Two\n");
    while (1)
        printf("Three\n");
    while (1)
        printf("Four\n");
    getchar();
}
```

One
Two
Three
Three
Three
...
Achieve Target 1

Note that the while loop for printing "Four" could be removed without affecting the program. The loop for printing "Three" iterates indefinitely anyway. We call this an infinite loop.

```c
#include <stdio.h>

void main() {
    if (1)    // One
        printf("One\n");
    if (1)    // Two
        printf("Two\n");
    while (1) // Three
        printf("Three\n");
        printf("Three\n");
        printf("Three\n");
        printf("Three\n");
        printf("Three\n");
        printf("...
");
    while (1) // Four
        printf("Four\n");
    getchar();
}
```

Remember it.

Any non-zero value, not just the value 1 in fact, would allow the execution in if and while statements.
Achieve Target 2

Study the following program.

```c
...  
    int count;
    count = 1;
    while (count) {
        printf("One\n");
        printf("Two\n");
    }
...  
```

The "One" and "Two" are printed repeatedly and indefinitely. Putting any number of statements between curly brackets makes the statements into one statement. We call that a **while body**.

Remember it.

When the execution reaches the end of the **while body**, the execution returns to the beginning where the red circle indicates. The decision on whether the execution of the while body happens when the execution reaches the red circle.

The variable `count` provides the value for the while statement execution control. The value was initialised to one and it never changed. So the while (compound) statement continued to execute indefinitely.
What happen if the value changes from non-zero to zero, or from zero to non-zero?

```c
...  
  int count;

  count = 1;
  while (count) {
    printf("One\n");
    printf("Two\n");
    count = 0;
  }
  printf("Finished\n");
  getchar();
  ...

The value of count changed to zero after printing "One" and "Two". The action caused the while statement not to execute anymore. So the execution flowed onto the printf statement writing "Finished". Note that the while loop executed only once.
The following program has a small change. The statement of assignment of zero to variable count is moved to a different place in the compound statement. But the program produced exactly the same output as the previous program.

```c
...  
int count;  
  count = 1;  
  while (count) {  
    count = 0;  
    printf("One\n");  
    printf("Two\n");  
  }  
  printf("Finished\n");  
getchar();  
...  
```

Remember it.

The value of `count` is changed to zero but the execution continued to print "One" and "Two". Only after printing "Two" is the changed value of `count` has an effect. You should know that although the value of `count` controls the execution of the `while` statement, the decision on the execution happens at the red circle only.

The value following the `while` keyword is the value that controls the execution of the `while` statement. This value is often referred as the `while` condition.
Achieve Target 3

Study the following program and then observe the output.

```c
... int count; count = 5; while (count) {
    printf("One\n");
    count = count - 1;
} printf("Finished\n"); ...
```

One
One
One
One
One
Finished
How many "One"s are printed?

```c
... int count;
    count = 5;
    while (count) {
        printf("One\n");
        count = count - 1;
    }

    printf("Finished\n");
...```

The "One" was printed five times, indicating that the while loop was executed five times.

Then the last printf statement printed "Finished".

Why is "One" printed five times? Not six times or four times?

The value of variable `count` clearly played an important role because it provides the value for the while statement execution control.
The following program may give us more insight by showing the changing values of count.

```c
int count;
count = 5;
while (count) {
    printf("count = %d\n", count);
    count = count - 1;
}

printf("Finished\n");
```

The value of count was initially five and each loop the value was reduced by 1. The while loop stopped or terminated at the red circle when count became zero.
Modify the program below so that it prints "Two" ten times.

```c
int count;
count = 5;
while (count) {
    printf("One\n");
    count = count - 1;
}

printf("Finished\n");
```

Work on the program to make it happen.
There are many different ways to write this program. The above shows two possible ways to make it happen. One program counts from 10 down to 0, and the second program counts from −10 up to 0.
You should now know how to control a while loop, and how many times would the while loop execute. We often call this counter-controlled while loops.

```c
int count;
count = 5;
while (count) {
    printf("count = %d\n", count);
    count = count - 1;
}
printf("Finished\n");
```

The counter is the variable that controls the while loop, and in this case, the variable count.
Study the following program and then observe the output.

```c
...  
  int count;

  count = 5;
  while (count > 0) {
    printf("%d\n", count);
    count = count - 1;
  }

  printf("Finished\n");
...  
```

The program behaves exactly the same way as the previous program.
Achieve Target 3A

The only change made is the while condition became \texttt{count > 0}. The while loop would continue as long as \texttt{count > 0} is non-zero.

```c
...  
  int count;  
  
  count = 5;  
  while (count > 0) {  
    printf("%d\n", count);  
    count = count - 1;  
  }  
  
  printf("Finished\n");  
...```

Learn it.

The expression \texttt{count > 0} is non-zero if \texttt{count} is greater than zero. Therefore the while loop continued when \texttt{count} is decreasing from 5 to 1. The while loop stopped when \texttt{count} became 0.
Modify the program below so that the first integer printed is 5 and last integer printed is –5, and each pair of subsequent numbers differs by 1. So the integers listed are 5 4 3 … -4 -5.

```c
... 
int count;

count = 5;
while (count > 0) {
    printf("%d\n", count);
    count = count - 1;
}

printf("Finished\n");
...
```

Work on the program to make it happen.
Solution

```c
#include <stdio.h>

void main() {
    int count;
    count = 5;
    while (count >= -5) {
        printf("%d\n", count);
        count = count - 1;
    }
    printf("Finished\n");
    getchar();
}
```

You can change the range of counter by changing the stopping condition of the while loop. Instead of stopping at 0, the while loop now stops beyond −5. So we have changed the operator to the greater and equal operator so that −5 is included.
Achieve Target 3B

Modify the program below again so that the following output is produced.

```c
...
int count;

count = 5;
while (count > 0) {
    printf("%d\n", count);
    count = count - 1;
}

printf("Finished\n");
...
```

5
3
1
-1
-3
-5
Finished

Work on the program to make it happen.

Observe the pattern and see how you can do this with the program.
#include <stdio.h>

void main() {
    int count;

    count = 5;
    while (count >= -5) {
        printf("%d\n", count);
        count = count - 2;
    }

    printf("Finished\n");
    getchar();
}
What is the **average** or the **mean** of the integers from **1 to 10**?
A quick answer is **5.5**.

```c
...  
void main() {
  int count;
  int sum;

  count = 1;
  sum = 0;
  while (count <= 10) {
    sum = sum + count;
    count = count + 1;
  }

  printf("Sum is %d and Average is %f\n", sum, (float)sum/10);
  getchar();
}
...  
```

**Sum is 55 and Average is 5.500000**

Let's check it with a program, which would find out the **sum of integers from 1 to 10**, and then **divide the sum by 10** for the **average**.
The variable \texttt{count} is the counter for this counter-controlled \texttt{while} loop. In this case, the counter is increasing rather than decreasing as in the previous programs.

```c
void main() {
  int count;
  int sum;
  count = 1;
  sum = 0;
  while (count <= 10) {
    sum = sum + count;
    count = count + 1;
  }
  printf("Sum is %d and Average is %f\n", sum, (float)sum/10);
  getchar();
}
```

The variable \texttt{sum} was initialised to \texttt{zero} and then at each loop the value of \texttt{count} was added to it.

So effectively the integers 1, 2, 3, .., 10 were added to \texttt{sum} in the 10 repetitions or iterations of the \texttt{while} loop. The while loop then stopped when \texttt{count} is more than 10 (or more precisely equalled to 11).
Achieve Target 4

The calculation of **average** is hidden within the `printf` statement, which is allowed as you learned in Perform Session 2.

```c
... void main() {
    int count;
    int sum;

    count = 1;
    sum = 0;
    while (count <= 10) {
        sum = sum + count;
        count = count + 1;
    }

    printf("Sum is %d and Average is %f\n", sum, (float)sum/10);
    getchar();
} ...
```

**Sum is 55 and Average is 5.500000**

The **(float)sum** is a new operation that **converts the type of sum's value to float.**

To a C program, the **type of a value/variable** is important because the type affects how operators behave. Making the `sum`'s value to `float` type would make the division a floating-point one rather than an integer one. Without it, the division became an integer one and the result would be 5.000000 instead.
The application of \texttt{(float)} to a variable is called \textit{type casting}.

```c
... 
void main() {
    int count;
    int sum;

    count = 1;
    sum = 0;
    while (count <= 10) {
        sum = sum + count;
        count = count + 1;
    }

    printf("Sum is \%d and Average is \%f\n", sum, (float)sum/10);
    getchar();
}
... 
```

Sum is 55 and Average is 5.500000

Of course we could cast a variable to integer type with \texttt{(int)} or cast into a character type with \texttt{(char)}. 

\textit{Remember it.}
Modify the above program so that it prints the sum of odd numbers from 1 to 11 (1 + 3 + 5 + ... + 11).

```c
void main() {
    int count;
    int sum;

    count = 1;
    sum = 0;
    while (count <= 10) {
        sum = sum + count;
        count = count + 1;
    }

    printf("Sum is %d and Average is %f\n", sum, (float)sum/10);
    getchar();
}
```

Sum is 55 and Average is 5.500000

Work on the program to make it happen.
Solution

```c
#include <stdio.h>

void main() {
    int count;
    int sum;

    count = 1;
    sum = 0;
    while (count <= 11) {
        sum = sum + count;
        count = count + 2;
    }

    printf("Sum is %d and Average is %f\n", sum, (float)sum/10);
    getchar();
}
```
The following program processes scores for a fixed 4 assignments. The program reports the average and the highest assignment score. Study the program.

```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();
}
```

Enter score for assignment 1: 80
Enter score for assignment 2: 60
Enter score for assignment 3: 50
Enter score for assignment 4: 70
Average score is 65.000000 and highest is 80.000000
Achieve Target 4A

The counter variable count no longer participates in other calculations. It controls the loop to repeat four times.

```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 \n%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();
...}
```

Enter score for assignment 1: 80
Enter score for assignment 2: 60
Enter score for assignment 3: 50
Enter score for assignment 4: 70
Average score is 65.000000 and highest is 80.000000
The really interesting feature is the detection of the highest score. The variable `highest` keeps 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.

```c
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();
...
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.

```c
while (count <= 4) {
    printf("Enter score for assignment\n%d: ", count);
    scanf("%f", &score);
    sum = sum + score;
    if (score > highest)
        highest = score;
    count = count + 1;
}
printf("Average score is %f and\nhighest is %f\n", sum/4, highest);
getchar();
...
```

Work on the program to make it happen.
Solution

```c
#include <stdio.h>

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

    count = 1;
    sum = 0;
    highest = 0;
    lowest = 100;

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

        if (score > highest)
            highest = score;
        if (score < lowest)
            lowest = score;

        count = count + 1;
    }

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

More exercises could be found in the exercise version
We have written a few programs on counter-controlled while loops. Let's study another way.

```c
... void main() {
    float data = 0;
    int count = 0;
    float sum = 0;
    printf("Enter numbers below (-1 to finish)\n");
    scanf("%f", &data);
    while (data != -1) {
        sum = sum + data;
        count++;
        scanf("%f", &data);
    }
    printf("Average is %f", sum/count);
    getchar();
}
```

The code for summation should be familiar to you.

The program calculates the average of the numbers entered repeatedly until -1 is entered.

Enter numbers below (-1 to finish)
5
4
3
2
-1
Average is 3.500000
Discuss the potential problem of designing a program that uses -1 as the sentinel value.

The while loop now stops when the value of data is -1, and the variable data holds the number entered from the input. So the while loop stops when the number entered is -1.

We refer to the number -1 as the sentinel value, which is a value that guards the execution of the while loop.
Achieve Target 5

It is essential to have two `scanf` statements for getting the input numbers.

```c
void main() {
    float data = 0;
    int count = 0;
    float sum = 0;
    printf("Enter numbers below (-1 to finish)\n");
    scanf("%f", &data);
    while (data != -1) {
        sum = sum + data;
        count++;
        scanf("%f", &data);
    }
    printf("Average is %f", sum/count);
    getchar();
}
```

The first `scanf` grabs the very first number for the while statement to operate correctly.

**Learn and Remember it.**

The second `scanf` then grabs the second and subsequent numbers.

This positioning of `scanf` statement is common for input-controlled while loops.
Achieve Target 5

Study the program that uses **counter-controlled while loop**.

```c

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

    printf("Enter 4 numbers below\n");
    while (count < 4) {
        scanf("%f", &data);
        sum = sum + data;
        count++;
    }
    printf("Average is %f", sum/4);
    getchar();
}
...  
```

Only one `scanf` statement is required. But only a fixed 4 numbers could be entered.
Two programs are required that allows the entering of Hang Seng Index (HSI) closing figure for each day, and then report the highest and the lowest closing index. In the first program, closing HSI for 5 days could be entered. In the second program, any number of closing HSI figures could be entered.

input-controlled while loop

```c
void main() {
    float data = 0;
    int count = 0;
    float sum = 0;
    printf("Enter numbers below (-1 to finish)\n");
    scanf("%f", &data);

    while (data != -1) {
        sum = sum + data;
        count++;
        scanf("%f", &data);
    }
    printf("Average is %f", sum/count);
    getchar();
}
```

counter-controlled while loop

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

    printf("Enter 4 numbers below\n");
    while (count < 4) {
        scanf("%f", &data);
        sum = sum + data;
        count++;
    }
    printf("Average is %f", sum/4);
    getchar();
}
```

Work on the program to make it happen.
Achieve Target 5

Solution

First Program

```c
#include <stdio.h>

void main() {
    float hsi = 0;
    int count = 1;
    float lowest = 9999999;
    float highest = -1;

    printf("Enter closing HSI for 5 days\n");

    while (count <= 5) {
        printf("Enter closing HSI for day %d: ", count);
        scanf("%f", &hsi);
        if (hsi < lowest)
            lowest = hsi;
        if (hsi > highest)
            highest = hsi;
        count++;
    }

    printf("Lowest HSI is %f and highest HSI is %f\n", lowest, highest);
    getchar();
}
```

Second Program

```c
#include <stdio.h>

void main() {

    float hsi = 0;
    float lowest = 9999999;
    float highest = -1;

    printf("Enter closing HSI (negative to finish)\n");
    scanf("%f", &hsi);

    while (hsi >= 0) {
        if (hsi < lowest) {
            lowest = hsi;
        }
        if (hsi > highest) {
            highest = hsi;
        }
        scanf("%f", &hsi);
    }

    printf("Lowest HSI is %f and highest HSI is %f\n", lowest, highest);
    getchar();
}
```
Achieve Target 6

The following program calculates the OCAS (assignment score) for MT258. It asks for the entering of 5 TMA scores and then performs the calculation according to the rule. The rule is that OCAS is the average of the best four TMA score.

```c
#include<stdio.h>

void main() {
    float data = 0;
    int count = 1;    /* counter initialisation */
    float sum = 0;
    float lowest = 100;

    printf("Enter 5 TMA scores below\n");

    while (count <= 5) { /* counter condition checking */
        printf("Enter TMA%d score: ", count);
        scanf("%f", &data);
        sum = sum + data;
        if (data < lowest)
            lowest = data;
        count++;
    } /* counter update */

    printf("OCAS is %f", (sum - lowest)/4);
    getchar();
}
You should find nothing new in the program. This is a counter-controlled loop and the three elements relating to the counter (variable count) are highlighted.

```c
#include<stdio.h>

void main() {
    float data = 0;
    int count = 1;    /* counter initialisation */
    float sum = 0;
    float lowest = 100;

    printf("Enter 5 TMA scores below\n");

    while (count <= 5) { /* counter condition checking */
        printf("Enter TMA%d score: ", count);
        scanf("%f", &data);
        sum = sum + data;
        if (data < lowest)
            lowest = data;
        count++;         /* counter update */
    }
    printf("OCAS is %f", (sum - lowest)/4);
    getchar();
}
```

Each of them serves a specified purpose in the loop, and they are initialisation, condition checking, and update.

Learn and Remember it.

You should find these three statements in all counter-controlled loops.
You will learn a new kind of loop that makes writing counter-controlled loop easier. Study the following program.

```c
#include<stdio.h>

void main() {

    float data = 0;
    int count;
    float sum = 0;
    float lowest = 100;

    printf("Enter 5 TMA scores below\n");

    for (count = 1; count <= 5; count++) {
        printf("Enter TMA%d score: ", count);
        scanf("%f", &data);
        sum = sum + data;
        if (data < lowest)
            lowest = data;
    }
    printf("OCAS is %f", (sum - lowest)/4);
    getchar();
}
```

The **for** loop structure gathers the three elements in one place, making it easier to read and write.
The two programs behave exactly the same. Study the program carefully to see the difference in coding.

Discuss the advantages of using for loops.
Achieve Target 6

The for loop statement has the following structure.

```
for (initialisation; condition checking; update)
```

Note that semicolons (;) separate the parts in the for loop statement.
Let's try to convert a counter-controlled while loop to a for loop. Convert the program using for loop and the for loop must contain the three elements, initialisation, condition checking, and update.

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

    printf("Enter 4 numbers below\n");
    while (count < 4) {
        scanf("%f", &data);
        sum = sum + data;
        count++;
    }
    printf("Average is %f", sum/4);
    getchar();
}
```

Work on the program to make it happen.
#include <stdio.h>

void main() {

    float data = 0;
    int count;
    float sum = 0;

    printf("Enter 4 numbers below\n");

    for (count = 0; count < 4; count++) {
        scanf("%f", &data);
        sum = sum + data;
    }

    printf("Average is %f", sum/4);
    getchar();
}
Achieve Target 7

In the last 6 targets, you have touched on the following ideas.

Ponder upon these ideas and remember what you have learned.
1. The **while** statement allows the **repeatedly** execution of the code in the **while** body. The **while** statement has a **condition** that controls the repeated execution. A **non-zero value** in the condition would allow the execution, and the **zero value** would disallow the execution.

2. The **while condition checking** happens when the execution reaches the **beginning of the while statement**. If the condition allows, then the while body is executed. **Once the while body execution is finished, the execution will return to the beginning of the while statement and go through another while condition checking. On the other hand, if the while condition failed, the execution will skip and jump the whole while body and resume from the next statement.**

3. A **while loop never stops** is called an **infinite loop**. It is often an erroneous behaviour and you should avoid it. **If a program doesn't stop and just hang, one possibility is that the program has a while loop. An easy way to test is to insert two printf statements into the program. One is placed before the while statement and another placed after the while body. If only the first printf statement is seen then the loop has not terminated.**
4. A counter-controlled while loop should have the following three features. A counter variable for keeping track of how many loops iterated so far; a while condition involving the counter variable; a statement that updates the counter variable. Counter-controlled while loop is most common in tasks in which we need to control the number of iterations.

5. A counter-controlled while loop can be re-written as a for loop. The advantage of a for loop structure is good readability. All three features of a counter-controlled while loop are gathered in the for statement in one place.
End of Session 4