Session 5
Loops 2
Designed by Andrew Kwok-Fai Lui and Maria Poon

All rights reserved. No part of this document may be reproduced, transmitted, or stored in a retrieval system, in any form or by any means, without permission in writing by the author.

Copyright © by Andrew Kwok-Fai Lui and Maria Poon 2002, 2003

Achieve Target 1

We begin this session with something new about while and for loops.

```c
#include <stdio.h>

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

The statement `break` forces the execution to leave the `while` (or `for`) loop immediately. The effect is the same as the `break` statement used in `switch` structure – to jump out of the structure. So "Three" is never printed because the `break` statement would force the execution of the loop to stop and leave the `while` loop.

Modify the above program so that it prints only "One" and "Four". **Work** on the program to make it happen.

The following program gives an equivalent `for` loop that runs indefinitely. Note the way how an infinite loop is written using a `for` structure.
#include <stdio.h>

void main() {
    printf("One\n");
    for (;;) {
        printf("Two\n");
        break;
        printf("Three\n");
    }
    printf("Four\n");
}
Achieve Target 1 Cont

Let's look at another program. This program should be familiar to you.

```c
#include <stdio.h>

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

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

The above program reads four numbers from the users and prints the average. A `while` loop is used to control the looping of 4 times with the help of the variable `count`.

We can rewrite the above program with a `break` statement and an infinite loop.

```c
... printf("Enter 4 numbers below\n");
while (1) {
    if (count >= 4)
        break;
    scanf("%f", &data);
    sum = sum + data;
    count++;
}
printf("Average is %f", sum/4);
getchar();
}
```

In the program the `break` statement is only conditionally executed. It is executed when the variable `count` is greater than or equal to 4. When the variable `count` is found to be the case, the `break` statement is executed and the loop stops.

The location of the `if - break` statement is important. The behaviour of the program changes if it is moved to a wrong place. The program below asks for 5 numbers instead of 4. Fix it by changing the statement in bold. **Work** on the program to make it happen.

```c
... printf("Enter 4 numbers below\n");
while (1) {
    scanf("%f", &data);
    sum = sum + data;
    if (count >= 4)
        break;
    count++;
}
printf("Average is %f", sum/4);
...
Achieve Target 2

The program below makes better use of the `break` statement to create non-trivial behaviour.

```c
#include <stdio.h>

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

    printf("Enter 4 numbers below (-1 to finish)\n");
    while (count < 4) {
        scanf("%f", &data);
        if (data == -1)
            break;
        sum = sum + data;
        count++;
    }

    if (count > 0)
        printf("Average is %f", sum/count);
    getchar();
}
```

The program allows users to enter up to four numbers. The user may, however, enter less than 4 numbers by terminating the input with the value –1. When –1 is entered, the `break` statement is executed.

Note that the statement for printing the average is now protected by an `if` statement. This is necessary to avoid the variable `sum` divided by zero, which is undefined in mathematics and would cause a `division-by-zero` error. Whenever you write code involving division, make sure you protect the program from division-by-zero error. Remember it.

Modify the program so that the input is terminated if the entered number is outside the range between 0 and 100. You may regard this program as an assignment average calculator. A value outside the range 0 and 100 is considered invalid. Work on the program to make it happen.
Achieve Target 2 Cont

We now need a program that calculates the average closing Hang Seng Index (HSI) of one trading week. There are 5 trading days per week. The user may terminate the input at any time by entering a negative number.

```c
#include <stdio.h>

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

    printf("Closing HSI Average for One Week (Enter < 0 to break)\n");

    while ( ) {
        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;
        }
    }

    if (count > 0)
        printf("Average is %f", sum/count);
    getchar();
}
```

You may base your work on the above skeleton. The skeleton provides a switch structure to provide clearer instructions when entering the HSI. Work on the program to make it happen.

Note that there are also break statements within the switch structure. Each break statement only breaks from one level of structure. So the break statements in the switch structure only break from switch structure, and not the higher level while structure. Remember it.
Achieve Target 3

We begin this session with another way to control the execution of for and while loops.

```
#include <stdio.h>

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

The statement `continue` forces the execution to return to the beginning of the loop structure. In the above program you would see the continued printing of "Two".

Modify the above program so that it prints only "Two" and "Three" indefinitely. Work on the program to make it happen.

The statement `continue` can be also used to control for loops. Modify the program above to use a for loop instead of a while loop. Work on the program to make it happen.
Achieve Target 3 Cont

We begin this session with another way to control the execution of for and while loops.

```c
#include <stdio.h>

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

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

    while (count < 4) {
        scanf("%f", &data);
        if (data < 0) {
            printf("Sorry, please enter positive number\n");
            continue;
        }
        sum = sum + data;
        count++;
    }
    if (count > 0)
        printf("Average is %f", sum/count);
    getchar();
}
```

The above program is the modified average calculator. The user is to enter 4 positive numbers and the average is calculated at the end. Instead of checking for the entering of -1 to terminate the input, this program checks for the validity of the entered data and asks the user to re-enter. The statement `continue` plays an important part in this design.

The if statement (in bold) checks if the entered data is negative. If this is the case, then a message is printed to the user and the continue statement will bring the execution back to the beginning of the while loop. The statements in blue are therefore not executed. It allows the re-entering of another number without updating the variable count and the addition of the sum.

With the above program, the user has to enter four positive number. Modify the program to allow the user to enter less than four positive number. As the programs used in the discussion of the break statement, the input of -1 signifies the terminating of input. All other negative numbers will be considered invalid and the user will be asked to re-enter. Work on the program to make it happen.
Achieve Target 4

The following program is the HSI average calculator discussed in a previous Target.

```
#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);

        sum = sum + data;
        count++;
    }
    if (count - holidaycount > 0)
        printf("Average is %f", sum/(count - holidaycount));
    getchar();
}
```

Sometimes we have public holidays and the stock exchanged is closed. The program should allow user to signify that the day is a holiday and there is no HSI index.

Modify the above program so that it can handle the entering of -1 as an indication that the day is a holiday and no index data is available. Note that the averaging should not include the holidays in the calculation. The variable `holidaycount` has been included in the skeleton to help you. Work on the program to make it happen.
Achieve Target 4 Cont

The following program is the same HSI average calculator but a for loop is used.

```c
#include <stdio.h>

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

    printf("Closing HSI Average for One Week (Enter -1 if holiday)\n");
    for (count = 0; count < 5; count++) {
        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);
        sum = sum + data;
    }
    if (count - holidaycount > 0)
        printf("Average is %f", sum/(count - holidaycount));
    getchar();
}
```

Repeat and modify the above program so that it can handle the entering of –1 as an indication that the day is a holiday and no index data is available. Note that the updating of the variable count is placed inside the `for` statement. You should consider whether the updating happen if a continue statement is executed. **Work** on the program to make it happen.
Achieve Target 5

We have written a few programs on loops now. Let's see something new.

```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();
} ...
```

Enter numbers below (-1 to finish)
5
4
3
2
-1

Average is 3.500000

Most of the programs use loop structures to control a fixed number of loops. For examples, a program read in four numbers and performed averaging. Another read in five HSI indexes and performed averaging. These programs would have a variable acting as a counter, which keeps track of how many loops have been performed so far. The counter would be part of the looping condition and take part in the control. We usually refer these loops as **counter-controlled loops**.

The above program uses a loop structure that does not depend on a counter. The program calculates the average of the numbers entered repeatedly until -1 is entered. The variable `data` holds the entered number. It is part of the while condition and so the variable `data` take part in the control of looping.

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. **Discuss** the potential problem of designing a program that uses -1 as the sentinel value.

It is essential to have **two** `scanf` statements for getting the input numbers. The first `scanf` grabs the very first number for the while statement to operate correctly. The second `scanf` then grabs the second and subsequent numbers. This positioning of `scanf` statement is a characteristic for **input-controlled while loops**. The first `scanf` should be just before the while loop structure, and the second `scanf` should be at the end of the while body. **Learn and Remember** it.
Achieve Target 5 Cont

Let's try to write a program that tells children whether the number entered is an even number or an odd number. The following code segment can be used to detect and report even or odd numbers.

```c
if (data % 2 == 0)
    printf("You have entered an even number\n");
else
    printf("You have entered an odd number\n");
```

Based on the input-controlled while loop discussed before, write the program for reporting even/odd numbers. The program should stop when –1 is entered. Note that the type of `data` should be `int` because we are dealing with even and odd numbers here. **Work** on the program to make it happen.
Achieve Target 6

We will look at input-controlled loops again with the following program.

```c
#include <stdio.h>
void main() {
    float data = 0;
    int count = 0;
    float sum = 0;

    printf("Enter numbers below (-1 to finish)\n");
    while (data != -1) {
        scanf("%f", &data);
        sum = sum + data;
        count++;
    }

    count = count - 1; /* -1 is not a number included in averaging */
    sum = sum - data; /* data should be -1 */

    printf("Average is %f", sum/count);
    getchar();
}
```

We discussed that a characteristic of input-controlled loop is the presence of two `scanf` statements. The first `scanf` is placed before the `while` loop for initialising the input variable. The second `scanf` is to read the subsequent input. Could we have one `scanf` statement instead?

The answer is yes or no, and in any case we must do it very carefully. The above program illustrates how it can be done. The singular `scanf` statement is now moved to the top of the `while` body. This is the only sensible place because the variable `data` must be assigned with entered number before it is added to the variable `sum` and so on.

There are two more things that we must be careful with.

First, we must make sure that the while loop is entered. The looping condition must be true before the while loop. In the program we make sure this to happen by assigning data to 0 (as long as it is not – 1).

Second, the values of variables count and sum are incorrect and need to be fixed. When user entered –1, the value –1 is added to the variable sum before the while loop terminated. The variable count was also incorrectly increased by one (-1 is not a number to be included in the averaging). So the first thing to do outside the while loop is to fix the values. Remember it.

On the other hand, it is not possible to rewrite the even/odd number program you wrote in the previous Target using one `scanf` statement, without the help of a `break` statement. Discuss the reason as why it cannot be done.

Our suggestion is to use two `scanf` statements in input-controlled loops.
Achieve Target 6 Cont

We will look at another new loop structure

```c
#include <stdio.h>

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

    printf("Enter numbers below (-1 to finish)\n");
    
    do {
        scanf("%f", &data);
        sum = sum + data;
        count++;
    } while (data != -1);

    count = count - 1; /* -1 is not a number included in averaging */
    sum = sum - data; /* data should be -1 */

    printf("Average is %f", sum/count);
    getchar();
}
```

This program is almost exactly the same as the previous program. The difference is the new do-while structure. In do-while structure, the looping condition checking is placed at the end. It makes no difference to the behaviour in the looping. When execution reaches the end, it returns to the top of a while loop anyway. So whether the looping condition checking is at the end of the loop or at the beginning of the loop makes no difference, except one thing. It affects the first loop.

In do-while, the loop structure is executed at least once. But while loop may not execute at all if the looping condition is initially false.

I do not regard do-while loops as particularly useful. All do-while loops can be rewritten in while loops anyway. We saw in the previous program that we could make while loop to execute at least once. Our suggestion is to learn do-while but try to write programs using while (and for) loops. Remember it.
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. The break statement forces the execution to leave the current loop structure. The execution then moves to the next statement after the structure. This statement is useful in providing alternative exits within a loop structure. Often for some complex loops, it is difficult to put every logic conditions in the looping condition. Using break statements could make the loop structure easier to write.

2. The continue statement forces the execution to return to the top of the loop structure. In a while loop, the looping condition checking is executed. In a for loop, the looping condition and the update part are both executed.

3. Both the break statement and the continue statement are replaceable by carefully crafted if statements. Both break and continue share the same effect of skipping the remaining loop body after the statement – one leaving the loop structure and the other returning to the top. An if structure could also be used to skip some part of the loop body and return the execution to the top of the loop. The use of break and continue, however, makes the code easier to read.

4. The use of sentinel values in input-controlled loops is a common practice in console-based applications. This allows users to signify that no more input would be entered. The choice of the sentinel value deserves some thoughts. We should always select a sentinel value that is not regarded as a probable input. We should also tell the user of the sentinel value in a prompting message.

5. The difference between do-while loop and while loop is that do-while executes at least one loop but while loop may not even execute one loop. However, while loop could be written in a way that executes at least one loop anyway. The do-while loops do not allow you to do more but they may make programs easier to read.