Session 4
Loops 1
Designed by Andrew Kwok-Fai Lui and Maria Poon

All rights reserved. No part of this note 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 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();
}
```

Output

```
Two
Four
Four
Four
Four
...
```

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. 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. Learn it.

We now want the program to give the following output, with "Three" printed repeatedly. Work on the program to make it happen.
Any non-zero value, not just the value 1 in fact, would allow the execution in `if` and `while` statements.  **Remember it.**

**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.  **Remember it.**

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?
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.
Achieve Target 2 Cont

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();
...
```

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. **Remember it.**

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.** **Remember it.**
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");
```

How many "One"s are printed? 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 at the top of the page so that it prints "Two" ten times. Work on the program to make it happens.

You have seen one way to control a `while` loop, and how many times would the while loop execute. The key is to control the while condition, such that it changes to zero when you want the loop to stop.
Achieve Target 3 Cont

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");
...  
```

Output

```
5
4
3
2
1
Finished
```

The above program behaves exactly the same way as the previous program. The only change made is
the while condition became `count > 0`. The while loop would continue as long as `count > 0` is non-zero. The expression `count > 0` is non-zero if `count` is greater than zero. Therefore the while loop continued when `count` is decreasing from 5 to 1. The while loop stopped when `count` became 0. **Learn it.**

Modify the above program 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. **Work** on the program to make it happens.

Modify the above program again so that the following output is produced. **Work** on the program to make it happens. Observe the pattern and see how you can do this with the program.

Output

```
5
3
1
-1
-3
-5
Finished
```

You could make further modifications to generate other patterns too. The locations of the red boxes in the above program indicate the places where you could make the modifications. You are encouraged to try out different values and see different patterns will emerge. **Work** on the program to make it happens.
Achieve Target 4

What is the average or the mean of the integers from 1 to 10? A quick answer is 5.5. 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.

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

```plaintext
Output
Sum is 55 and Average is 5.500000
```

The variable `count` is increasing rather than decreasing and the `while` condition is the result of comparing if `count` is less than or equal to 10. The while loop stopped when count is more than 10 (or more precisely 11).

What are interesting are the actions performed in each loop. In a loop the variable `count` takes on a different value, increased by one each time. The variable `sum` is added with the variable `count`. Over the 10 loops, integers 1, 2, 3, ..., 10 were added to variable `sum`.

The sum and the average printed are calculated in the usual way. The calculation of average is hidden within the `printf` statement, which is allowed as you learned in Perform Session 2.

Modify the above program so that it prints the sum of odd numbers from 1 to 11 (1 + 3 + 5 + ... + 11). Work on the program to make it happen. After you have successfully written one, try to write another program that achieves the same. There are many ways to write this program.

Let's sidetrack a little for the moment. 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. Work on the program and try it out and see it happen.

The application of `(float)` to a variable is called type casting. Of course we could cast a variable to integer type with `(int)` or cast into a character type with `(char)`. Remember it.
Achieve Target 4 Cont

The following program evaluates the factorial of 10. The factorial of 10 is \(1 \times 2 \times 3 \times \ldots \times 10\). You could find the function factorial on your calculator labelled as \(n!\).

```c
#include <stdio.h>
void main() {
    int count;
    int fac;
    fac = 1; /* must initialised to 1 for the multiplication to work */
    count = 2; /* so count starts from 2 */
    while (count <= 10) {
        fac = fac * count;
        count = count + 1;
    }
    printf("Factorial of 10 is %d\n", fac);
    getchar();
}
```

Output

```
Factorial of 10 is 3628800
```

Note that the variables are initialised differently. The variable \(\text{fac}\) will stores the temporary value of the multiplication sequence and it is initialised to 1. For the multiplication statement to work, the variable \(\text{fac}\) must not be 0 (otherwise the multiplication always results in 0). Because \(\text{fac}\) is already initialised to 1, the multiplication can start at 2 (instead of 1) and so count is initialised to 2.

Modify the program so that it calculates the factorial of 12 instead. **Work** on the program to make it happen.

It is useful to identify some common features found in while loops. The first feature (in blue colour) is the **while condition**. It controls whether to continue looping. The second feature (in red colour) is **initialisation**, which perform the necessary action before going into the while loop. The third feature (in green) is **while body**, which is executed repeatedly in the loops. We will discuss the second and the third features in more details.

More than one statement may play the role of initialisation or the while body. The statements for initialisation are however executed only once before the while loop. On the other hand, the statements for the while body are executed repeatedly. Note that almost all while body will **update** the while condition. For example in the above program, the increment of the variable \(\text{count}\) will eventually make the while condition to false. **Remember** it.
Achieve Target 5

It is time to introduce another way to write loops – the for loop structure. The following program again calculates the factorial of 10.

```c
#include <stdio.h>
void main() {
    int count;
    int fac;
    fac = 1;
    count = 2;
    for ( ; count <= 10; ) {
        fac = fac * count;
        count = count + 1;
    }
    printf("Factorial of 10 is %d\n", fac);
    getchar();
}
```

The while statement is replaced by a for statement in the above program. This is the only modification made. Note that the looping condition is now placed between two semicolons. Otherwise, the program has exactly the same structure and the same effect as before.

Again modify the program so that it calculates the factorial of 30 instead. Work on the program to make it happen.

The for statement has something more to offer. A for statement actually has three parts. The following diagram illustrates the purpose of the parts.

```
for (initialisation ; looping condition; update)
  for body;
```

Two semicolons separate the three parts. The familiar looping condition is the second part. This is equivalent to the role of the while condition – non-zero means looping and zero means stopping. The initialisation part is executed once and before the for loop begins.

The for body is the same as the while body in that it is executed repeatedly. The update part is also executed as part of the for body. Specifically, it is executed after the for body.

The only significance thing about for loop is that it allows us to move some of the statements for initialisation and those inside the body together in the for statement. Remember it.
Achieve Target 5 Cont

The following program shows the moving of a statement into the initialisation part of the for statement.

```c
... void main() {  
    int count;  
    int fac;  
    fac = 1;  
    for (count = 2; count <= 10; ) {  
        fac = fac * count;  
        count = count + 1;  
    }  
    printf("Factorial of 10 is %d\n", fac);  
    getchar();  
}  
...
```

The update part remains empty, which is alright. The initialisation part now has one statement that assigns 2 to count. It can take more than one statement as illustrated in the following program.

```c
... void main() {  
    int count;  
    int fac;  
    for (fac = 1, count = 2; count <= 10; ) {  
        fac = fac * count;  
        count = count + 1;  
    }  
    printf("Factorial of 10 is %d\n", fac);  
    getchar();  
}  
...
```

A comma separates the two items in the initialisation parts. Again this modification has no effect on the program.

We could move a statement of the for body into the update part of the for statement, as illustrated in the following program.

```c
... void main() {  
    int count;  
    int fac;  
    for (fac = 1, count = 2; count <= 10; count = count + 1) {  
        fac = fac * count;  
    }  
    printf("Factorial of 10 is %d\n", fac);  
    getchar();  
}  
...
```

Again this modification has no effect on the program. We could also have multiple items in the update part. Modify the program so that the remaining for body is moved into the update part. **Work** on the program to make it happen. Note that the ordering of items is significant.
Achieve Target 6

The following program gives a clearer illustration on the execution order of various parts of a for loop. The execution order is important to writing programs.

```c
void main() {
    int count;
    for (printf("For init\n"), count = 0; printf("For cond\n"), count < 5;
         printf("For update\n"), count++) {
        printf("For body count = %d\n", count);
    }
    getchar();
}
```

We have also changed the statement

```c
    count = count + 1;
```

to

```c
    count++
```

These two forms are equivalent, but the latter one is simpler to write and to read.

We can put `printf` statements as an item in the `for` statement. Basically we could put any statement as an item in any part of the `for` statement. A `printf` statement is just another statement but we could use it to illustrate how the `for` loop works.

We could see that the initialisation part is executed once and before anything else. Condition checking follows it. **Study** the output and reaffirm your understanding of for loops.
Achieve Target 6 Cont

All for loops can be re-written as while loops, and vice versa. The following program is a rewritten form of the previous program using while loops.

```c
void main() {
    int count;
    printf("While init\n");
    count = 0;
    while (printf("While cond\n"), count < 5) {
        printf("While body count = %d\n", count);
        printf("While update\n"), count++;
    }
    getchar();
}
```

There is now no clear distinction between update and the while body. In fact, update is explicitly part of the while body. The comma (,) in the second statement is used in the same way as in the for statement. The use is to allow two expressions in a statement.

Though for loops do not offer anything special, there is an added value in using for loops. The for loops allows the gathering of those statements relevant to the looping condition in one place. The following program, for example, gathers the statement concerning the initialisation, checking, and updating of the variable count in one place. The advantage is that it makes the program easier to read.

```c
void main() {
    int count;
    int fac;
    fac = 1;
    for (count = 2; count <= 10; count++) {
        fac = fac * count;
    }
    printf("Factorial of 10 is %d\n", fac);
    getchar();
}
```
Achieve Target 6 Cont

The following program reads in four numbers from the user, and evaluates the average of the four numbers.

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

Except that the program now reads data within the loop, there is nothing new introduced in it.

Rewrite the program to use a for loop. Select the appropriate items into the for loop so that the program is easier to read. **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. 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 while loop should have the following three features. An initialisation part that executes once before the loop begins. A looping condition part that control whether to continue looping. The while body, containing the update part that changes the looping condition.

5. A 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 while loop may be gathered in the for statement in one place. But for good readability, only those relevant to the looping condition are gathered there.