COMP1511: Iterations using `while`

Session 2, 2018

Iterations: `while` Statements

- We often need to execute code (statements) many times.
- `if` statements only allow us to execute or not execute code. in other words they allow us to execute code 0 or 1 times
- `while` statements allow us to execute code **0 or more times**
- Like `if`, `while` statements have a controlling expression

- `while` statements execute their body ***until*** the *controlling expression* is ***false***,
  - in other words, a set of statements inside a `while` statement are executed ***while*** the *controlling expression* is ***true***

```
int loop_counter = 0;

loop_counter = 0;
while (loop_counter < 5) {

    printf("*");

    loop_counter = loop_counter + 1;

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

Initialisation

Controlling expression

Body of the while

Update (counter)

`while` Statements

- C has other looping constructs - but **`while`** is all you need
- **`for`** loops can be a little more concise/convenient, we'll see them later - for now use **`while`**
- Often use a loop ***counter*** variable to count loop repetitions (iterations)
- Can then have a **`while`** loop execute **`n`** times.

```c
#include <stdio.h>

int main (void) {

    // read an integer n
    // print n asterisks
    int loop_counter, n;

    printf("How many asterisks? ");
    scanf("%d", &n);

    loop_counter = 0;
    while (loop_counter < n) {

        printf("*");

        loop_counter = loop_counter + 1;
    }

    printf("\n");

    return 0;
}
```

`while` Loop - Loop Counter Pattern

Here is the programming **pattern** for a `while` that executes **n** times:

```
loop_counter = 0;
while (loop_counter < n) {
    //
    // one or more statements
    // the loop needs to execute
    //

    loop_counter = loop_counter + 1;
}
```

`while` - Termination

- Can control **termination** (**stopping**) of `while` loops in many ways.
- Easy to write `while` loops that **do not terminate !**
- Be careful, make sure that *controlling expression* eventually becomes ***false***
- Often a ***sentinel variable*** is used to stop a `while` loop,
  value of this *sentinel variable* is **updated** in every iteration (cycle).

```c
// Example : use of sentinel value (here, mark != -1)
// Alternatively, we can also say (mark >= 0)

#include <stdio.h>

int main (void) {
    int mark;

    printf("Enter mark? ");
    scanf("%d", &mark);

    while ( mark != -1 ) {

        if(mark >= 50) {
            printf("You Passed!\n");
        }
        else {
            printf("Sorry, you Failed!\n");
        }

        // Get next mark
        printf("Enter mark? ");
        scanf("%d", &mark);
    }
    return 0;
}
```

Initialisation

Controlling expression

Body of the while

Update (sentinel variable)

`while` Loop - Sentinel Variable Pattern

```
printf("Enter mark? ");
scanf("%d", &mark);

while ( mark != -1 ) {

    if(mark >= 50) {
        printf("You Passed!\n");
    }
    else {
        printf("Sorry, you Failed!\n");
    }

    // Get next mark
    printf("Enter mark? ");
    scanf("%d", &mark);
}
```

Initialisation Sentinel variable

Check Sentinel Variable

Body of the while

Update Sentinel variable

`while` Termination : `scanf` example (1/2)

- **scanf** uses a format string like **printf**. Notice **&** before the variable name.
- scanf **returns number** of items **successfully read**
- Consider the following `scanf` statement:

```
int answer, noRead;
printf("Enter the answer: ");
noRead = scanf("%d", &answer);
```

➔ If we input an integer, `scanf` will return **1**.

➔ If we input a string, say "john", it results in zero item successfully read, so `scanf` returns **0**.

- If we try to **read beyond the end** of an input stream (end of file), `scanf` also **returns zero**.
- We can indicate **end of input** stream by pressing **CTRL + D**

`while` Termination : `scanf` example (2/2)

```c
int main (void) {
    int mark, noRead;

    printf("Enter mark? ");
    noRead = scanf("%d", &mark);

    while ( noRead == 1 ) {

        if(mark >= 50) {
            printf("You Passed!\n");
        }
        else {
            printf("Sorry, you Failed!\n");
        }

        // Get next mark
        printf("Enter mark? ");
        noRead = scanf("%d", &mark);
    }
    printf("\n Bye ... \n");
    return 0;
}
```

Typically `scanf` will return zero in this program if,

- a user enters a value other than integer, OR
- an input stream reaches its end, so nothing to read

Nested `while` Loops

- Often need to nest while loops.
- Need a separate loop counter variable for each nested loop.

```c
int main (void) {

    // print a square of 10x10 asterisks
    int i, j;
    i = 0;
    while (i < 10) {

        j = 0;
        while (j < 10) {
            printf("* ");
            j = j + 1;
        }

        printf("\n");
        i = i + 1;
    }


    return 0;
}
```

*Body of **outer** loop*

*Body of **inner** loop*

Nested `while` Loops: Example

```c
#include <stdio.h>

int main (void) {

    int i, j;
    i = 0;
    while (i < 3) {

        j = 0;
        while (j <= 2) {
            printf("i=%d , j=%d \n", i, j);
            j = j + 1;
        }

        printf("\n");
        i = i + 1;
    }

    return 0;
}
```

*Body of **outer** loop*

*Body of **inner** loop*

Output:

```
i=0 , j=0
i=0 , j=1
i=0 , j=2

i=1 , j=0
i=1 , j=1
i=1 , j=2

i=2 , j=0
i=2 , j=1
i=2 , j=2
```

Linux: Output Redirection

- We can **redirect** standard output to a terminal from a program (or command) to another file.
- By appending “ `> myfile` ” to the command, we can redirect standard output for that command to the file named “`myfile`”

For example,

```
% ls   > myfile

% date  > myfile2
```

Linux: Input Redirection

- The standard input of a command/program can be **redirected** from a given file (in place of a keyboard).
- By appending " `< myfile` " to the command, we can redirect standard input from the file named "`myfile`"

For example, the following program "`findGrade`" will read input from the file named "`myfile`", and **not** from a keyboard.

```
% findGrade  <  myfile
```

`printf`: output formatting with printf

Integer

`%wd` means

- `w` is width in integer and
- `d` is conversion specification

Double

`%w.clf` means

- `w` is width in double,
- `c` specifies the number of digits after decimal point and
- `lf` is conversion specification

Value	Placeholder	Output (□ means blank)
-10	%d	-10
	%2d	-10
	%4d	□-10
	%-4d	-10□
10	%04d	0010
49.76	%.3lf	49.760
	%.1lf	49.7
	%.-10.2lf	49.76□□□□□
	%10.2lf	□□□□□ 49.76
	%10.3e	□4.976e+01

Arithmetic and Assignment Operators

- `++` is increment operator, it increases the integer value by one.
  Say x is 15, `x++` will increase x to 16.
- `--` is decrement operator, it decreases the integer value by one.
  Say x is 15, `x--` will decrease x to 14.

- `+=` is “Add AND assignment operator”. It adds the right operand to the left operand and assign the result to the left operand.
  For example `val = val + 5` is same as `val += 5`
- `-=` is “Subtract AND assignment operator”. It subtracts the right operand from the left operand and assigns the result to the left operand.
  For example `val = val - 5` is same as `val -= 5`