

Foundations of C Programming (Structured Programming)

- Loops

Outline

- *while* loop
- *do-while* loop
- *for* loop
- The *break* statement
- The *continue* statement
- The *goto* statement

Loops

- In our daily life, some actions have been repeated.
 - E.g.,
 - Eat 10 bites of an apple
 - Eat an apple until it is finished
- In a C program, we can also describe repeated actions



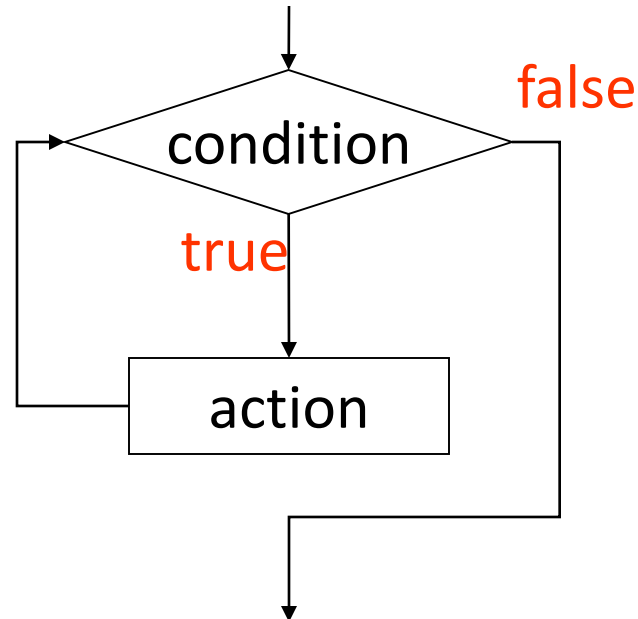
The *while* Loop

Syntax:

```
while (<condition>)  
    <action>;
```

or

```
while (<condition>) {  
    <action>;  
}
```



- If **condition** is true then execute **action**
- Repeat this process until **condition** evaluates to **false**
- **action** is either a single statement or a group of statements within a pair of curly brackets

An Example

```
int n = 10;
while(n > 1) {
    printf("%d ", n);
    n--;
}
```

- What is the final value of *n*
- How many times "*n--*" is executed?
- What is the output of this program?
- What is this program's flow chart?

An Example

- Compute factorial of n ($n!$)
- First step, we need to work out the algorithm for this computation.

$$- 1! = 1$$

$$- 2! = 2 * 1 = 2 * 1!$$

$$- 3! = 3 * 2 * 1 = 3 * 2!$$

$$- 4! = 4 * 3 * 2 * 1 = 4 * 3!$$

$$- \dots$$

$$- n! = n * (n - 1) * \dots * 1 = n * (n - 1)!$$

An Example

Compute factorial of n ($n!$)

```
int number, factorial, counter;
printf("Enter a positive integer:");
scanf("%d", &number);

factorial = 1;  // initialization
counter = 1;

while(counter <= number){
    factorial = factorial * counter;
    counter++; //counter = counter + 1;
}
printf("The factorial of %d is %d.",
       number, factorial);
```

Class Exercise

- Compute 2^n
- First step, we need to work out the algorithm for this computation.
 - $2^0 = 1$
 - $2^1 = 2 * 2^0$
 - $2^2 = 2 * 2^1$
 - $2^3 = 2 * 2^2$
 - ...
 - $2^n = 2 * 2^{n-1}$

Can you write a program to compute 2^n using *while*

- Compute factorial of n ($n!$)
- First step, we need to work out the algorithm for this computation.

$$- 1! = 1$$

$$- 2! = 2 * 1!$$

$$- 3! = 3 * 2!$$

$$- 4! = 4 * 3!$$

$$- \dots$$

$$- n! = n * (n - 1)!$$

- Compute 2^n
- First step, we need to work out the algorithm for this computation.

$$- 2^0 = 1$$

$$- 2^1 = 2 * 2^0$$

$$- 2^2 = 2 * 2^1$$

$$- 2^3 = 2 * 2^2$$

$$- \dots$$

$$- 2^n = 2 * 2^{n-1}$$

An Example

```
int value; // input value
int max = 0; // maximum value
printf("Enter a positive integer (-1 to stop):");
scanf("%d", &value);
while(value != -1){
    if(value > max)
        max = value;
    printf("Enter a positive integer (-1 to stop):");
    scanf("%d", &value);
}
printf("The maximum value is %d", max);
```

What does this program do?

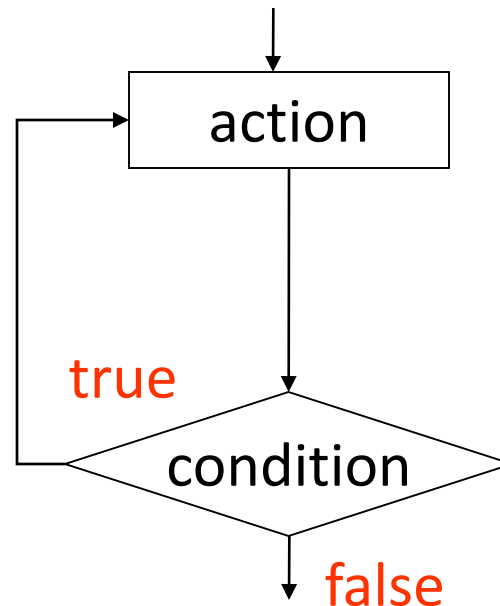
The *do-while* Loop

Syntax:

```
do
    <action>;
while (<condition>) ;

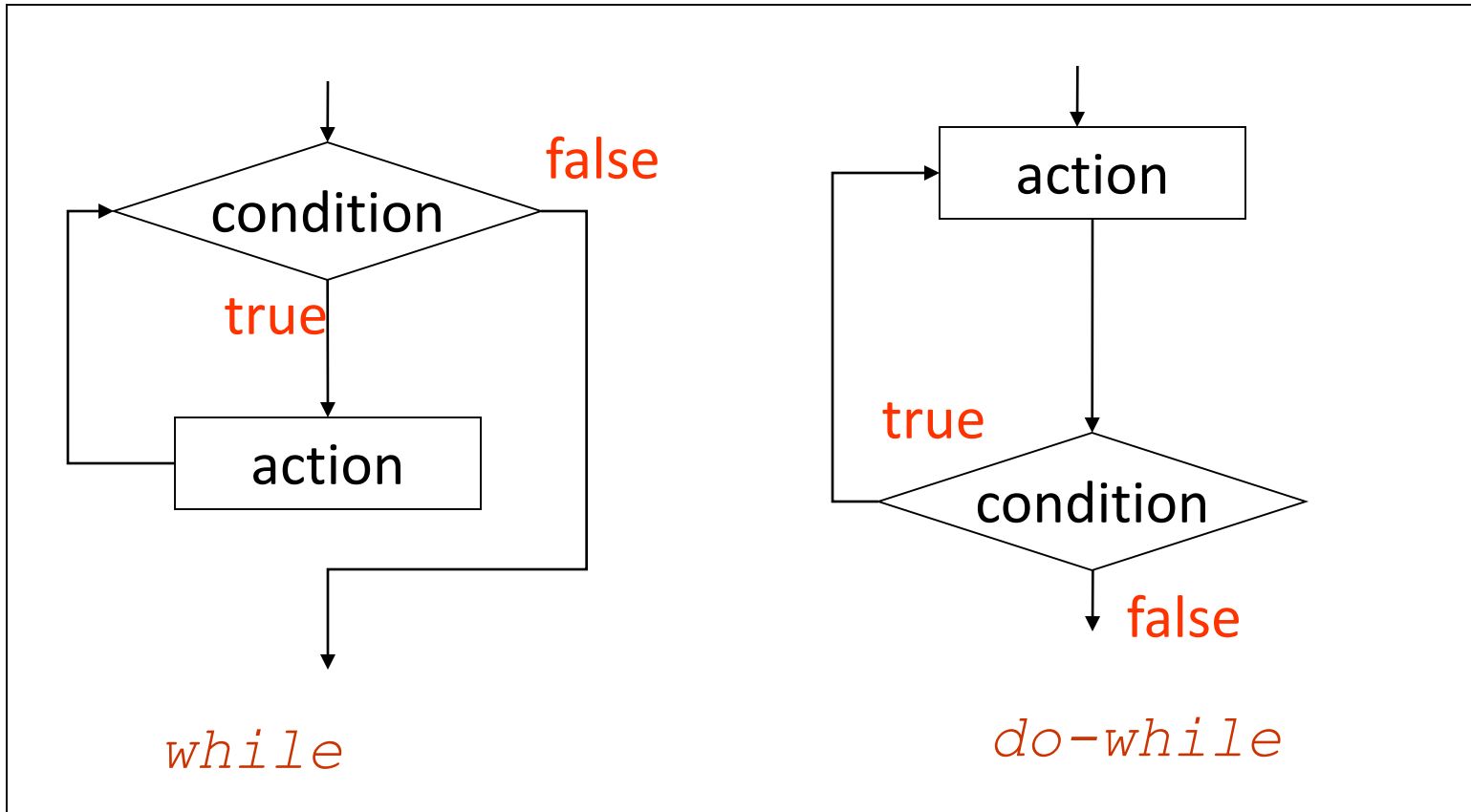
or

do {
    <action>;
} while (<condition>) ;
```



- First execute the **action**, then check the **condition** of the loop
- **action** is either a single statement or a group of statements within a pair of curly brackets

Compare while and do-while



What are the differences?

Compare while and do-while

- *while* loop
 - First check the condition of the loop
 - then execute the body of the loop
- *do-while* loop
 - First execute the body of the loop
 - Then check the condition of the loop
 - the body of the loop is executed at least once

The $n!$ Example

```
int number, factorial, counter;

printf("Enter a positive integer:");
scanf("%d", &number);
factorial = 1;  // initialization
counter = 1;

do{
    factorial *= counter;
    counter++;
}while(counter <= number);

printf("The factorial of %d is %d.",
       number, factorial);
```

Any difference between this program and the one which uses **while**?

After-Class Exercise

- Compute 2^n
- First step, we need to work out the algorithm for this computation.
 - $2^0 = 1$
 - $2^1 = 2 * 2^0$
 - $2^2 = 2 * 2^1$
 - $2^3 = 2 * 2^2$
 - ...
 - $2^n = 2 * 2^{n-1}$

Can you write a program to compute 2^n using *do-while*

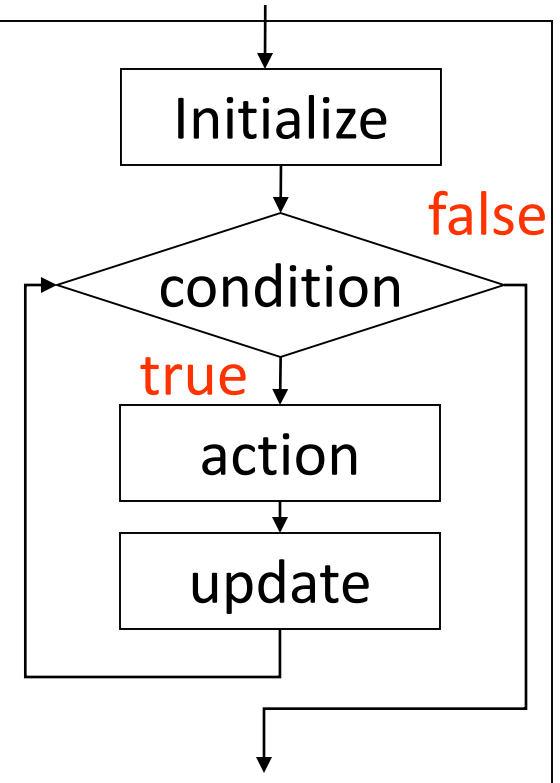
The *for* Loop

Syntax:

```
for (<initialize>; <condition>; <update>)  
    <action>.
```

or

```
for (<initialize>; <condition>; <update>) {  
    <action>  
}
```



- Initialize first
- while **condition** is **true**, execute **action** and execute **update**
- **Initialization, condition and update** can be empty

The $n!$ Example

```
int number, factorial, i;

printf("Enter a positive integer:");
scanf("%d", &number);
factorial = 1;  // initialization

for(i = 1; i <= number; i++)
    factorial *= i;

printf("The factorial of %d is %d.",
       number, factorial);
```

After-Class Exercise

- Compute 2^n
- First step, we need to work out the algorithm for this computation.
 - $2^0 = 1$
 - $2^1 = 2 * 2^0$
 - $2^2 = 2 * 2^1$
 - $2^3 = 2 * 2^2$
 - ...
 - $2^n = 2 * 2^{n-1}$

Can you write a program to compute 2^n using *for*?

Attentions in Loops

- Make sure there is a statement that will eventually **stop** the loop.
 - **Infinite loop** == loop that never stops or stops after unreasonable unexpected huge number of loops.

```
int i = 1;
int number = 100;
int sum = 0;
while (i <= number) {
    sum = sum + i;
    i--;
}
printf("the sum of integers from 1 to
      100 is %d", sum);
```

Infinite Loops

- The following format can cause the infinite loops if no special statement is used to terminate the loop
 - while (1)
 - for (; ;)
- Wrong conditions can cause infinite loops
 - E.g.

```
scanf ("%d", &n) ;  
while (n = 10) { // always true  
    .....  
}
```

Attentions in Loops

- Make sure to initialize loop counters correctly.
 - **Off-by-one** == the number of times that a loop is executed is one more time or one less.

```
int i = 1;
int number = 100;
int sum = 0;
while (i < number) {
    sum = sum + i;
    i++;
}
printf("the sum of integers from 1 to
      100 is %d", sum);
```

Which Loop to Use?

- *for* loop
 - for calculations that are repeated a fixed number of times
 - controlled by a variable that is changed by an equal amount (usually 1) during each iteration
- *while* loop
 - The number of iterations depends on a condition which could be changed during execution.
- *do-while* loop
 - The code segment is always executed at least once.

Examples

```
for (i = 1; i <= 10; i++)  
    printf("*****\n");
```

```
i = 1;  
while (i <= 10) {  
    printf("*****\n");  
    i++;  
}
```

```
i = 1;  
do {  
    printf("*****\n");  
    i++;  
} while (i <= 10);
```

Compare these three segments, which one is better?

Examples

```
char reply;
printf("*****\n");
printf("continue? (y/n)");
scanf("%c%c", &reply);
while(reply != 'n') {
    printf("*****\n");
    printf("continue? (y/n)");
    scanf("%c%c", &reply);
}
```

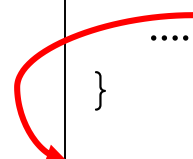
Compare these two segments,
which one is better?

```
char reply;
do{
    printf("*****\n");
    printf("continue? (y/n)");
    scanf("%c%c", &reply);
} while(reply != 'n');
```


Stop the Loop

- There are two ways to stop the loop
 - Normal way: check the conditions in the *for*, *while* and *do-while*, if the condition is false, stop the loop.
 - Forced way: Use *break* statement
 - When the *break* statement is executed, the loop statement terminates immediately.
 - The execution continues with the statement following the loop statement.

```
while (C) {  
    .....  
    break;  
    .....  
}  
.....
```



Examples

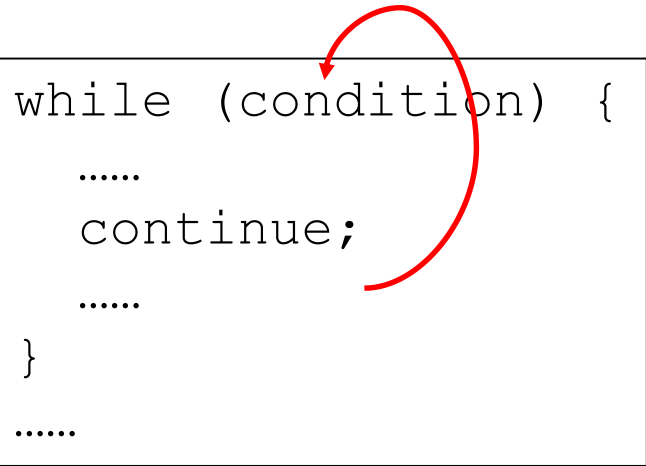
```
sum = 0;
for (i = 1; i <= 100; i++) {
    sum = sum + i;
    if (sum >= 1000)
        break;
}
printf("i = %d, sum = %d", i, sum);
```

```
sum = 0;
for (i = 1; i <= 100; i++)
    sum = sum + i;
printf("i = %d, sum = %d", i, sum);
```

Compare these two programs

The *continue* Statement

- The *continue* command terminates the *current* iteration (i.e., ignore the rest statements) and starts the *next* iteration.



```
while (condition) {  
    .....  
    continue;  
    .....  
}  
.....
```

Examples

```
sum = 0;
for (i = 1; i <= 100; i++) {
    if (i % 2 == 0)
        continue;
    sum = sum + i;
}
printf("i = %d, sum = %d", i, sum);
```

```
sum = 0;
for (i = 1; i <= 100; i++)
    sum = sum + i;
printf("i = %d, sum = %d", i, sum);
```

Compare these two programs

Examples

```
sum = 0;
for (i = 1; i <= 100; i++) {
    if (i % 2 == 0)
        continue;
    sum = sum + i;
}
printf("i = %d, sum = %d", i, sum);
```

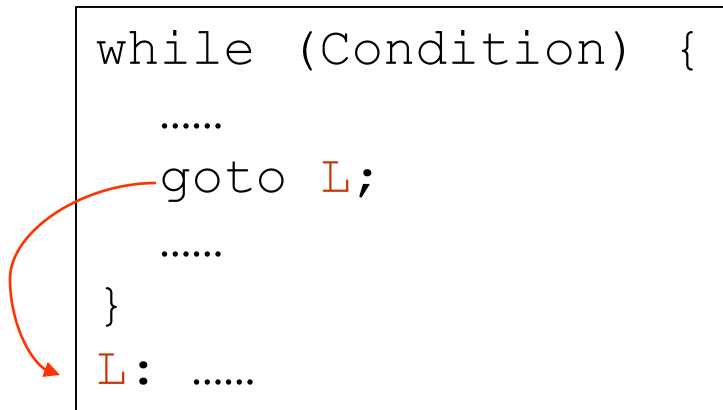
What if we change *continue* to *break*?

An Example

```
int mycard = 3;
int guess;
for(;;) // infinite loop, we can also use while(true)
{
    printf("Guess my card:");
    scanf("%d",&guess);
    if(guess == mycard){
        printf("Good guess!\n");
        break;    // get out of the infinite loop
    }
    else
        printf("Try again.\n");
}
```

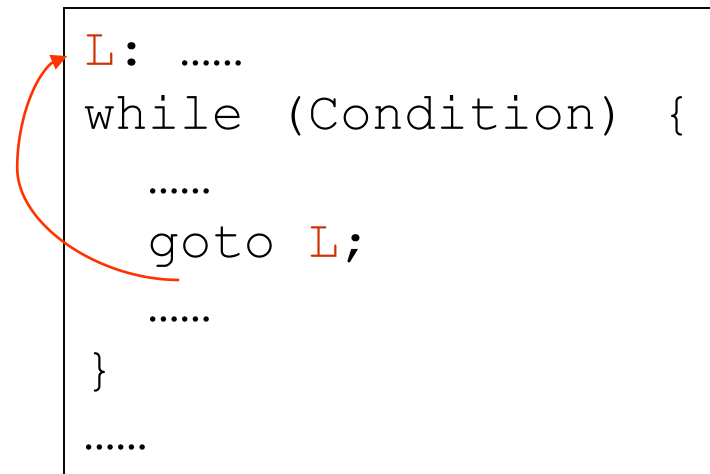
The *goto* Statement

- It performs a one-way jump to another line of code.
- The jumped-to locations are usually identified using **labels**



```
while (Condition) {  
    .....  
    goto L;  
    .....  
}  
L: .....
```

A diagram illustrating a `goto` statement. A red arrow originates from the `goto L;` line and points to the label `L:` at the bottom of the code block, indicating a forward jump.



```
L: .....  
while (Condition) {  
    .....  
    goto L;  
    .....  
}  
.....
```

A diagram illustrating a `goto` statement. A red arrow originates from the `goto L;` line and points back to the label `L:` at the top of the code block, indicating a backward jump.

The *goto* Statement

- The **goto** statement is often combined with an *if* statement to cause a conditional transfer of control.

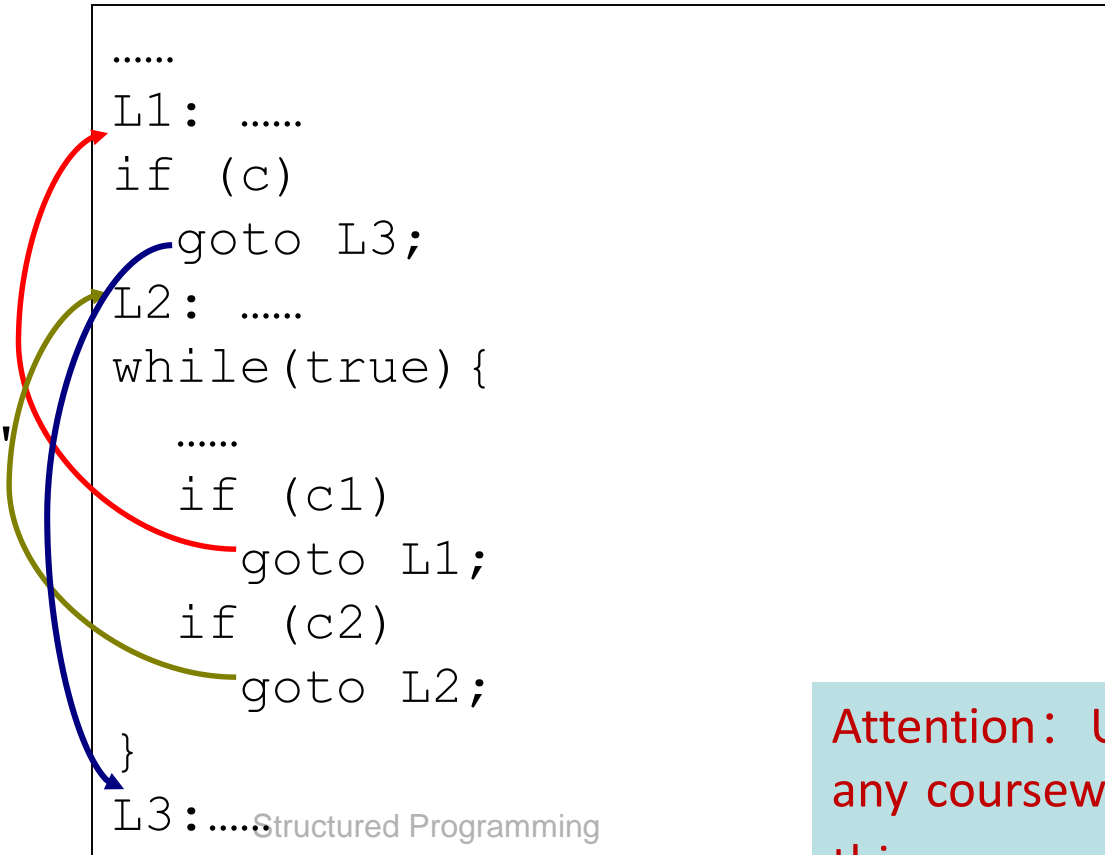
```
int mycard = 3;
int guess;
for(;;){ // infinite loop, we can also use while(true)
    printf("Guess my card:");
    scanf("%d",&guess);
    if(guess == mycard){
        printf("Good guess!\n");
        goto L;    // get out of the infinite loop
    }
    else
        printf("Try again.\n");
}
L: printf("Guess successfully!\n")
```


The *goto* Statement

- Use of *goto* statements results in "spaghetti code" that is difficult to read and maintain
- It is suggested that **NO *goto*** statements are used in programs

"spaghetti"

```
.....  
L1: .....  
  if (c)  
    goto L3;  
L2: .....  
  while(true) {  
    .....  
    if (c1)  
      goto L1;  
    if (c2)  
      goto L2;  
  }  
L3: .....
```



Attention: Use of *goto* in any coursework or exam in this course will be given 0

Nested Loops

- Nested loops are loops within loops.
- Nested loops are similar in principle to nested *if* and *if-else* statements.

```
int row;      // Outer loop counter
int col;      // Inner loop counter

for(row = 1; row <= 10; row++){
    for(col = 1; col <= 10; col++){
        printf("%d ", row * col);
        printf("\n");
    }
}
```

1. How many times "printf("%d",row*col);" is executed?
2. How many times "printf("\n");" is executed?
3. What is the output of this program?

Nested Loops

Output:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 73 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

1. How many times `printf("%d",row*col);` is executed? **100**
2. How many times `printf("\n");` is executed? **10**

Class Exercise

- How to use nested loops to produce the following output?

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

Summary

- Loops in a program
 - while
 - do-while
 - for
- Pay attention to the difference between *continue* statement and *break* statement
- Loops can be nested