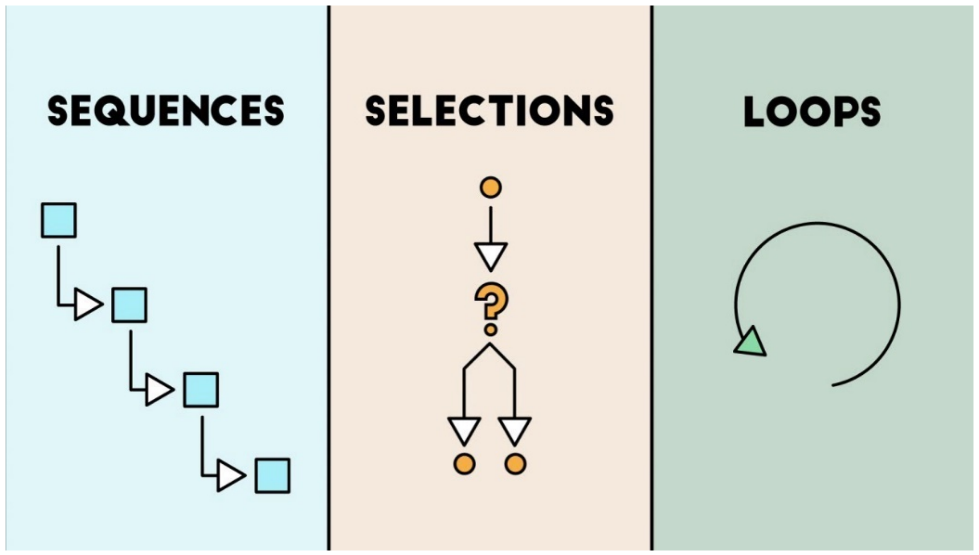
Control Structures (Part 2)

This topic seems to be simple (not much to remember)
but it can be hard (in terms of how you apply and code, and solve problems)

Control Structures

What is Control?

- Which piece of code to be executed next?



Three Basic types:

- Sequence
- Conditional – If (…) do this
- Iteration – Repeat this until (…)

Outline

- **`while`** loop
- **`for`** loop
- Nested loops
- Interrupting the control flow in a loop with **`break`** and **`continue`**

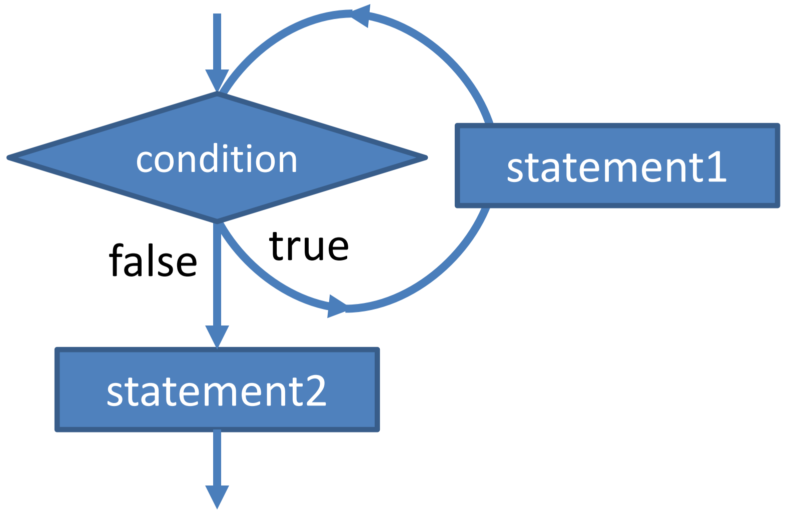
Loop

- An **`if-else`** statement allows some statements to be executed zero or one times.

- A loop statement allows some statements to be executed repeatedly **zero or more** times.

1. `while` statement (syntax)

```
while ( condition )
    statement1 ;
statement2 ;
```



- Repeat **statement1** as long as **condition** is **true**
  - *Like a "jail", one can't leave until* condition *becomes false*
- After **condition** == **false**, exit the loop and move to the statement after the loop, i.e., **statement2**.

1.1. while statement (Example #1)

```
int i ;

i = 1 ;    // 1. initialize loop control variable

// A simple loop that iterates 5 times
while ( i <= 5 ) {            // 2. test condition
    printf( "%d\n" , i ); // 3. loop body
    i++ ;                     // 4. update loop control variable
}

printf( "Lastly, i = %d\n" , i );
```

```
1
2
3
4
5
Lastly, i = 6
```

1.2. Key components of a loop

```c
int i ;

i = 1 ;

// A simple loop that iterates 5 times
while ( i <= 5 ){
     printf( "%d\n" , i );
    i++ ;
}
```

1. "Loop variable" initialization

Assign a value to the variable to be used in the loop condition, such that the loop condition true initially

2. Loop condition

- When this condition is true, the loop body is executed.
- Usually controlled by a variable

3. Loop body

Statements to be repeated

4. Change of loop condition

To stop the loop, we need to make the loop condition false. This can usually be done by changing the loop variable.
You should never omit it... unless...

1.3. `while` statement (Example #2)

- Given the following task (as an example):

  Step 1. Ask the user for a number.

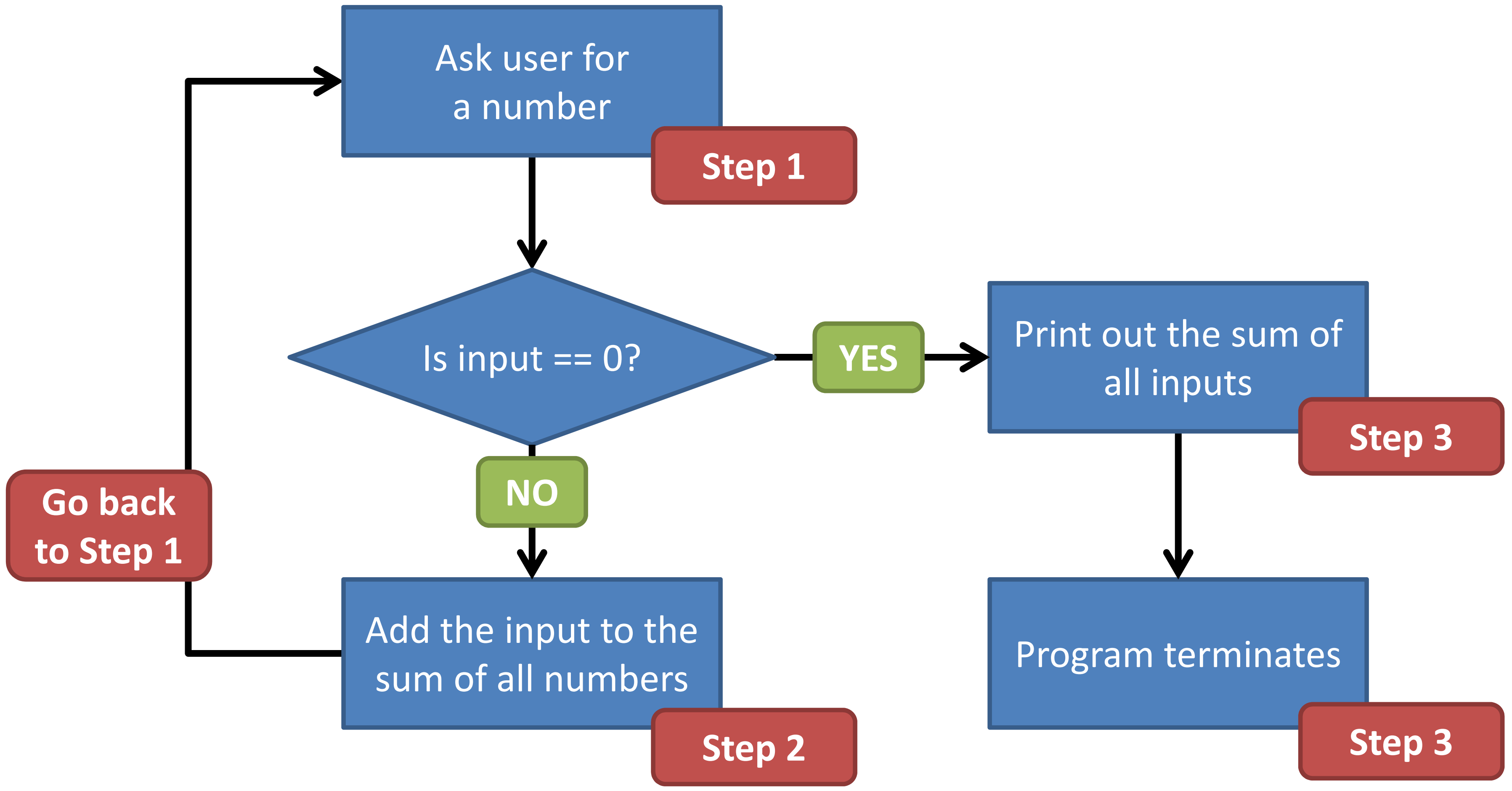
  Step 2. If the input value is not zero, add it to the sum of all previous inputs and go back to Step 1.

  Step 3. If the input is zero, print the sum of all inputs and terminate the program.

- How can we write it in a while-loop program?

1.3. `while` statement (Example #2)

- Transforming steps 1 – 3 into a flow chart:



1.3. `while` statement (Example #2)

```c
int input , sum = 0 ; // To store input value and their sum
int getZero = 0 ;      // To control the loop:
                       //  1 => stop loop; 0 => continue loop
while ( getZero == 0 )
{
    printf( "Input: " );
    scanf( "%d" , &input );

    if ( input == 0 )
        getZero = 1 ;
    else
        sum += input ;
}

printf( "Sum = %d\n" , sum );
```

```
Input: 1
Input: 3
Input: 5
Input: 7
Input: 0
Sum = 16
```

1.3. `while` statement (Example #2)

```c
int input , sum = 0 ; // To store input v
int getZero = 0 ;       // To control the l
                        //  1 => stop loop;
while ( getZero == 0 )
{
    printf( "Input: " );
    scanf( "%d" , &input );

    if ( input == 0 )
        getZero = 1 ;
    else
        sum = sum + input ;
}

printf( "Sum = %d\n" , sum );
```

Loop variable initialization

Loop condition

Loop condition update (conditionally)

Usually, all key components of a loop are included.

1.4. Infinite Loop

RULE: can’t leave a loop until condition changes!

Hence…

- A loop that never stops. e.g.,

```
while ( 1 )
   printf( "Hello!\n" );
```

- Usually introduced by mistakes
- What could happen when a program runs into an infinite loop?

1.4.1. Common mistakes that result in infinite loops

- A condition that is always true

```
while ( a > -10 || a < 10 ) {
 …
}
```

- Fail or forget to update/modify the value of the loop variable inside the loop

```
i = 0 ;
while ( i <= 5 ) {
  printf( "i = %d\n" , i );
}
```

  - In this example, **`i`** is always 0.

1.4.1. Common mistakes that result in infinite loops

- Using **=** instead of **==** as equality operator

```
while ( a = 1 ) {
  ...
}
```

  – Variable **a** is assigned 1 and the whole expression is always evaluated to 1, and 1 means true.

- Placing '**;**' after the condition of a **while** loop

```
while ( a != 0 ) ;
{
   …
}
```

; represents an *empty statement*. That is

```
while ( a != 0 ) ;
```

is interpreted the same as

```
while ( a != 0 ) {
}
```

Outline

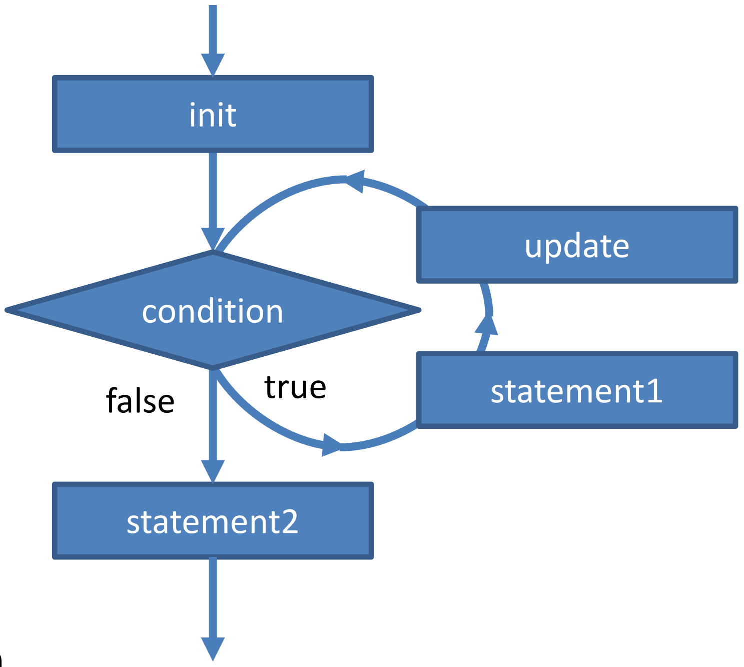
- ~~**`while`** loop~~
- **`for`** loop
- Nested loops
- Interrupting the control flow in a loop with **`break`** and **`continue`**

2. for statement (Syntax)

- The **initialization** (init) statement
  - Execute once and before the **condition** statement

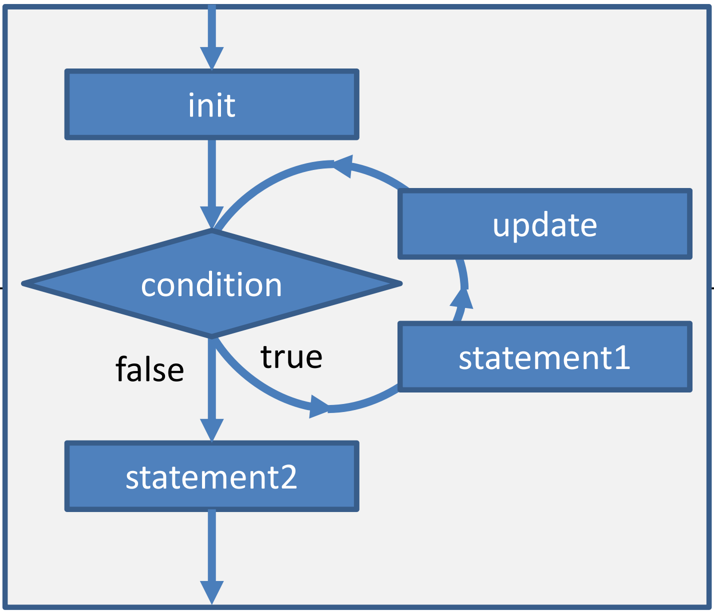
```
for ( init ; condition ; update )
    statement1 ;
statement2 ;
```

- The **condition** statement
  - Same as while-loop: test the condition at the beginning of each iteration
- The loop body (**statement1**)
  - Repeat until **condition** becomes false
- The **update** statement
  - Executed after **statement1** in each iteration
  - Usually for updating the loop condition



2.1. for statement (Example #3)

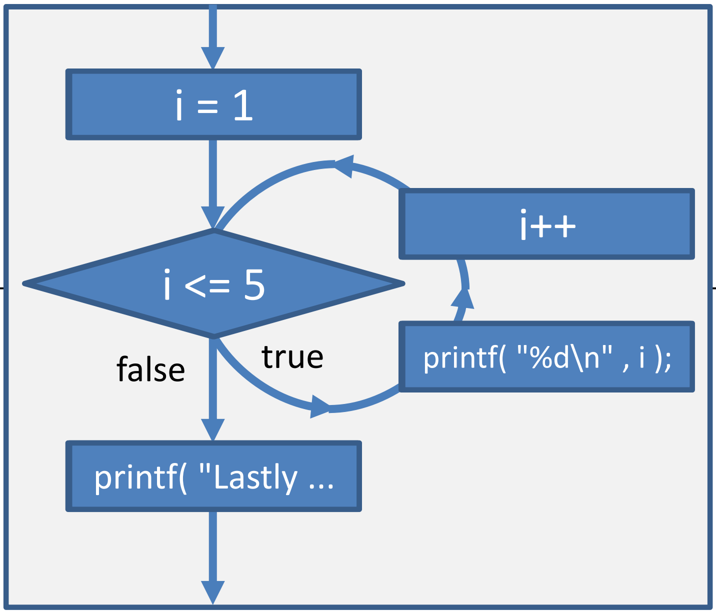
```
int i ;

// A simple loop that iterates 5 times
for ( i = 1 ; i <= 5 ; i++ )
    printf( "%d\n" , i );

printf( "Lastly, i = %d\n" , i );
```



2.1. for statement (Example #3)

```
int i ;

// A simple loop that iterates 5 times
for ( i = 1 ; i <= 5 ; i++ )
    printf( "%d\n" , i );

printf( "Lastly, i = %d\n" , i );
```

```
1
2
3
4
5
Lastly, i = 6
```



2.1. `for-loop` vs. `while-loop`

while-loop version

```c
int i ;

i = 1 ;
while ( i <= 5 ) {
     printf( "%d\n" , i );
     i++ ;
}
```

for-loop version

```c
int i ;

for ( i = 1 ; i <= 5 ; i++ ) {
     printf( "%d\n" , i );
}
```

Comparison

2.1. `for-loop` vs. `while-loop`

1. **Counter-controlled loop:**
   - The number of repetitions can be *known* before the loop body starts; just repeat the loop on each element in a preset sequence
   - Usually implemented using `for-loop`

2. **Sentinel-controlled loop:**
   - The number of repetitions is *NOT known* before the loop body starts. For example, a *sentinel value* (e.g., –1, different from normal data)
   - Usually implemented using `while-loop`

2.1. `for-loop` vs. `while-loop`

```
sum = 0.0
REPEAT N times
   ASK user for next student's height
   sum += height
END of REPEAT
average = sum / N
```

Which type?

```
sum   = count = 0
time  = get current time
WHILE time < Canteen A closing time
        height = get height of next guy
        sum    += height
        count += 1
        time = get current time
END WHILE
```

Which type?

2.1. `for-loop` vs. `while-loop`

- They are "equivalent" in terms of capability:
  - For any task you can accomplish with one of these loop structures, you can also accomplish it with the other loop structure.

- But in general, `for`-loop is more expressive for tasks to be repeated in a finite number of times, where we know the number of iterations when the loop starts

2.2. for statement (Example #4)

```c
int i , num , N ;
scanf( "%d" , &N );
// i changes from 0 to N-1
for ( i = 0 ; i < N ; i++ ) {
    num = N – i ;
    printf( "%d\n" , num );
}
```

```c
int i , num , N ;
scanf( "%d" , &N );
// i changes from 1 to N
for ( i = 1 ; i <= N ; i++ ) {
    num = N – i + 1 ;
    printf( "%d\n" , num );
}
```

```c
int i , num , N ;
scanf( "%d" , &N );
// i changes from N to 1
for ( i = N ; i >= 1 ; i-- ) {
    num = i ;
    printf( "%d\n" , num );
}
```

Different ways to print out the numbers from N to 1 using a for loop.

* The numbers we want to generate in a loop can usually be expressed in terms of the **loop variable.**
* **“One more or one less iteration”** can kill your program!!!

2.2. for statement (Example #4)

```
int i , num , N ;
scanf( "%d" , &N );
// i changes from 0 to N-1
for ( i = 0 ; i < N ; i++ ) {
    num = N – i ;
    printf( "%d\n" , num );
}
```

```
int i , num , N ;
scanf( "%d" , &N );
// i changes from 1 to N
for ( i = 1 ; i <= N ; i++ ) {
    num = N – i + 1 ;
    printf( "%d\n" , num );
}
```

```
int i , num , N ;
scanf( "%d" , &N );
// i changes from N to 1
for ( i = N ; i >= 1 ; i-- ) {
    printf( "%d\n" , i );
}
```

Different ways to print out the numbers from N to 1 using a for loop.

* The numbers we want to generate in a loop can usually be expressed in terms of the **loop variable.**
* **"One more or one less iteration"** can kill your program!!!

2.3. for-loop - skip some components

```
int main( void )
{
  int i = 0 ;
  for ( i = 0 ; i < 3 ; i++ ) printf( "loop 1: i=%d\n" , i   );
  for (       ; i < 6 ; i++ ) printf( "loop 2: i=%d\n" , i   );
  for (       ; i < 9 ;     ) printf( "loop 3: i=%d\n" , i++ );

  for ( ; ; ) printf( "loop 4: i=%d\n" , i++ );
  return 0 ;
}
```

- You may skip some component parts in a for loop.
- But missing all of them will result in **"an infinite loop." (when no condition)**

How if…

```
int main( void )
{
  int i ;

  // can't compile!!!
  for ( i = 0 , i < 3 , i++ ) printf( "loop 1: i=%d\n" , i   );
}
```

- Semicolon ; and comma , have very different meanings in C language
- You must ***use semicolon to end a statement*** and also ***to separate the components in a for loop***

```
// comma as a separator to help initialize multiple variables
for ( i = 0 , j = 0 ; i < 3 ; i++ )
```

Tips on Planning to Write a Loop

- Before you write a loop, please make sure you figure out:
  - Before the Loop: What should be done before the loop?
    - You almost always need to initialize the looping variable(s)
    - You may need to initialize the variables that persists through your repetition
  - Inside the Loop: What should be done repeatedly? And how should the loop variable change?
  - After the Loop: What should be done after all repeats are finished?

Tips: Debugging a Loop

- Your loop will be wrong if you are confused with what you should do Before/Inside/After the loop
- When you write a loop or debug a loop, you have at least these TWO things to check:
  - Structurally, it should have loop variable initialization, looping condition and loop condition update
  - Logically, you should make sure statements before/inside/after the loop is in the right place

Outline

- ~~**`while`** loop~~
- ~~**`for`** loop~~
- Nested loops
- Interrupting the control flow in a loop with **`break`** and **`continue`**

3. Nested loops – A loop inside another loop

```
int i , j ;
for ( i = 1 ; i <= 3 ; i++ )
{
    for ( j = 1 ; j <= 4 ; j++ )
    {
        printf( "%d %d\n" , i , j );
    }
}
```

```
1 1
1 2
     i = 1
1 3
1 4
2 1
2 2
     i = 2
2 3
2 4
3 1
3 2
     i = 3
3 3
3 4
```

- The whole loop can be considered as only ONE statement.
- For each outer loop iteration, the inner loop iterates 4 times.

3.1. Nested loops (Example #5)

- Objective: To print a ***multiplication table*** in the following format:

9 rows

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
......
9 18 27 36 45 54 63 72 81 90
```

- What are the things being repeated?
  - There are 9 rows
  - Each row contains 10 numbers

3.1. Nested loops (Example #5)

```c
int i , j ;
for ( i = 1 ; i <= 9 ; i++ )      // 9 rows
{
    for ( j = 1 ; j <= 10 ; j++ ) // 10 numbers per row
    {
        printf( "%d " , ______________ ); // your code here
    }
    printf( "\n" );    // Newline appears only once per row
}
```

- What expression, in terms of `i` and `j`, will yield the numbers we need?

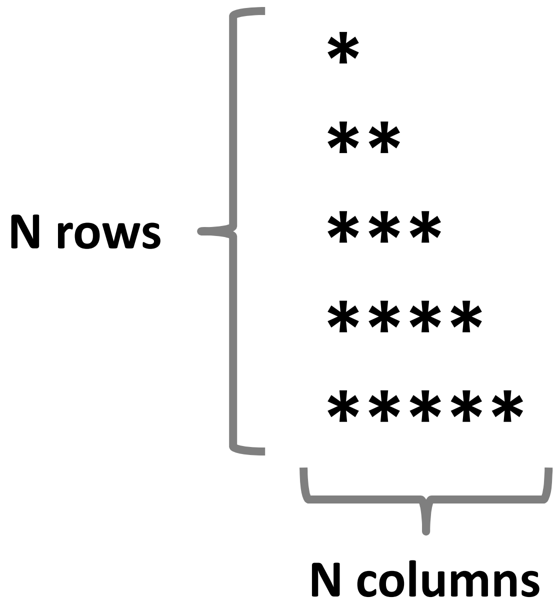
3.1. Nested loops (Example #5)

```
int i , j ;
for ( i = 1 ; i <= 9 ; i++ )       // 9 rows
{
    for ( j = 1 ; j <= 10 ; j++ ) // 10 numbers per row
    {
        printf( "%d " , i * j );
    }
    printf( "\n" );    // Newline appears only once per row
}
```

- Often, the numbers we want to generate inside a loop (or nested loops) can be expressed using the loop variables.

3.2. Nested loops (Example #6)

- Objective: Given a positive integer N, print out a triangle in the following format (e.g., when N = 5):



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

Note:
This is a rather basic and simple example.
See past exam paper!!!

3.2. Nested loops (Example #6)

```c
int i , j , N ;

printf( "N = ? " );
scanf( "%d" , &N );

for ( i = 1 ; i <= N ; i++ )           // N rows
{
    for ( j = 1 ; j <= i ; j++ )       // row i has i stars
        printf( "*" );
    printf( "\n" );
}
```

Four levels of skills

#1 Understand the flow: trace and understand code

#2 Analysis: Given a problem, carefully think and design the logic of the loops

#3 Apply: Transform the logic appropriately into for/while (with "good programming style")

#4 Test: think about all possible consequences and evaluate your code accordingly

Practice!

Practice!!

Practice!!!

Note:
for #1 & #2: you may read textbooks and see more examples

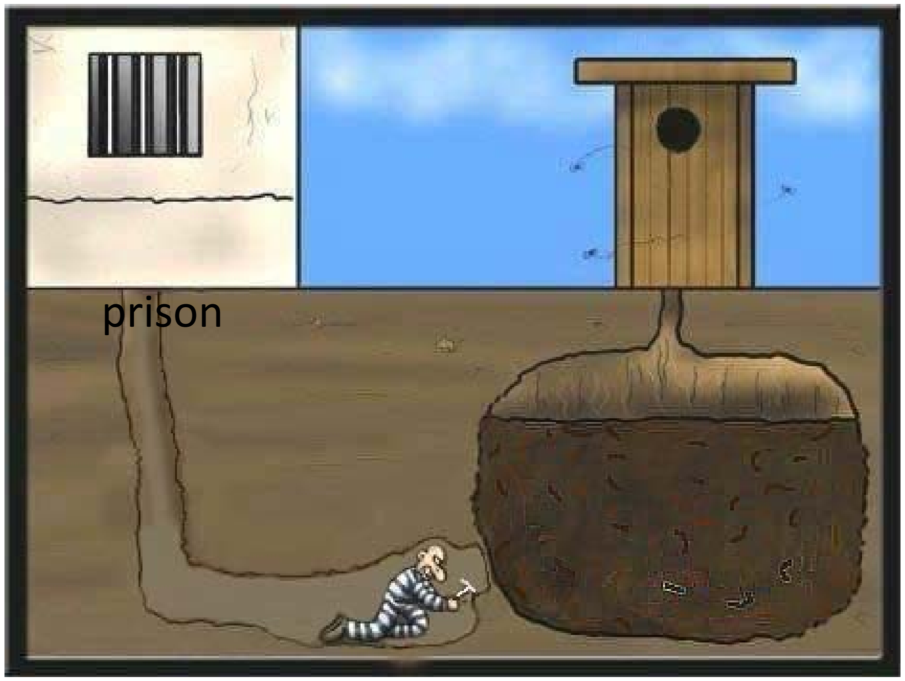
Outline

- ~~**`while`** loop~~
- ~~**`for`** loop~~
- ~~Nested loops~~
- Interrupting the control flow in a loop with **`break`** and **`continue`**

Appendix

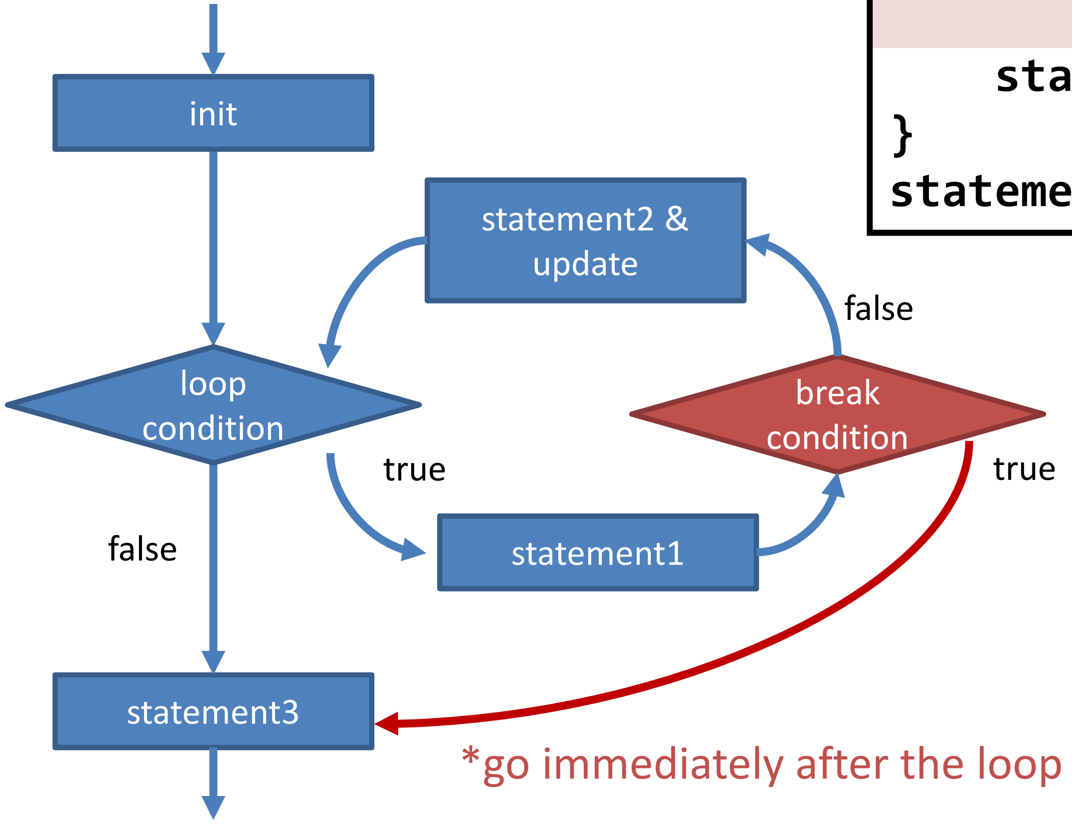
- **`break`** statement
- **`continue`** statement

Note: they **interrupt** the normal control flow in a loop



What is **break**?

```
for ( init ; loopCondition ; update )
{
    statement1 ;
    if ( breakCondition )
        break ;
    statement2 ;
}
statement3 ;
```



*go immediately after the loop

Early Exit in a Loop

The **break** statement, when executed, causes the program to **leave the closest enclosing loop** immediately.

Examples of **break** statement

```c
int input , sum = 0 ;  // To store input value and their sum

while ( 1 )           // This is an infinite loop
{
    printf( "Input: " );
    scanf( "%d" , &input );

    if ( input == 0 )
        break ;      // break the loop when input value == 0

    sum = sum + input ;
}

printf( "Sum = %d\n" , sum );

```

Rewriting the example on Page 8 using `while ( 1 )`.

Examples of **break** statement

- Using break to stop a “for loop”

```
int i ;

for ( i = 0 ; i < 10 ; i++ )
{
    printf( "here\n" );
    if ( i == 3 )
        break ;

    printf( "%d\n" , i );
}

printf( "Bye!\n" );
```

What is the output?

```
here
0
here
1
here
2
here
Bye!
```

A common mistake with **break**

- A break

```c
int i ;

for ( i = 0 ; i < 10 ; i++ )
{
    printf( "here\n" );
    // if ( i == 3 )
        break ;

    printf( "%d\n" , i );
}

printf( "Bye!\n" );
```

What is meaning of the code?

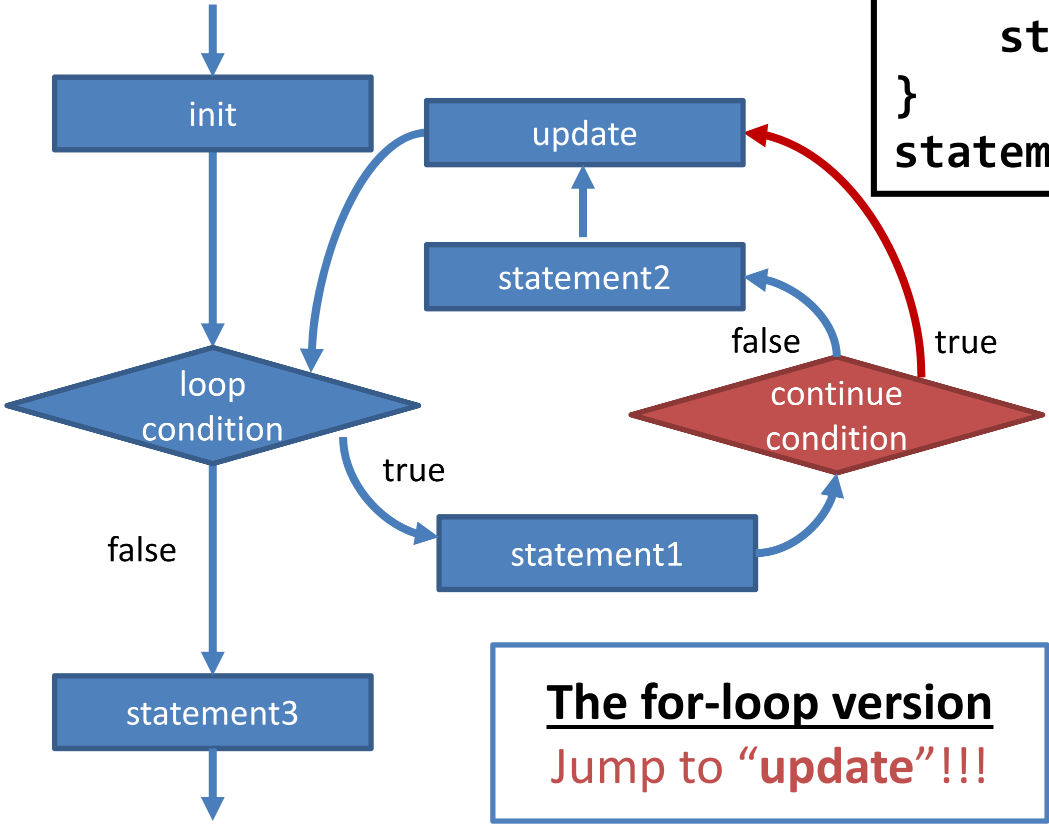
```c
int i ;

i = 0 ;
printf( "here\n" );

printf( "Bye!\n" );
```

Usually, we use **break inside an if statement**

What is continue?

```
for ( init ; loopCondition ; update )
{
    statement1 ;
    if ( continueCondition )
        continue ;
    statement2 ;
}
statement3 ;
```

PAY HIGH ATTENTION!!!!



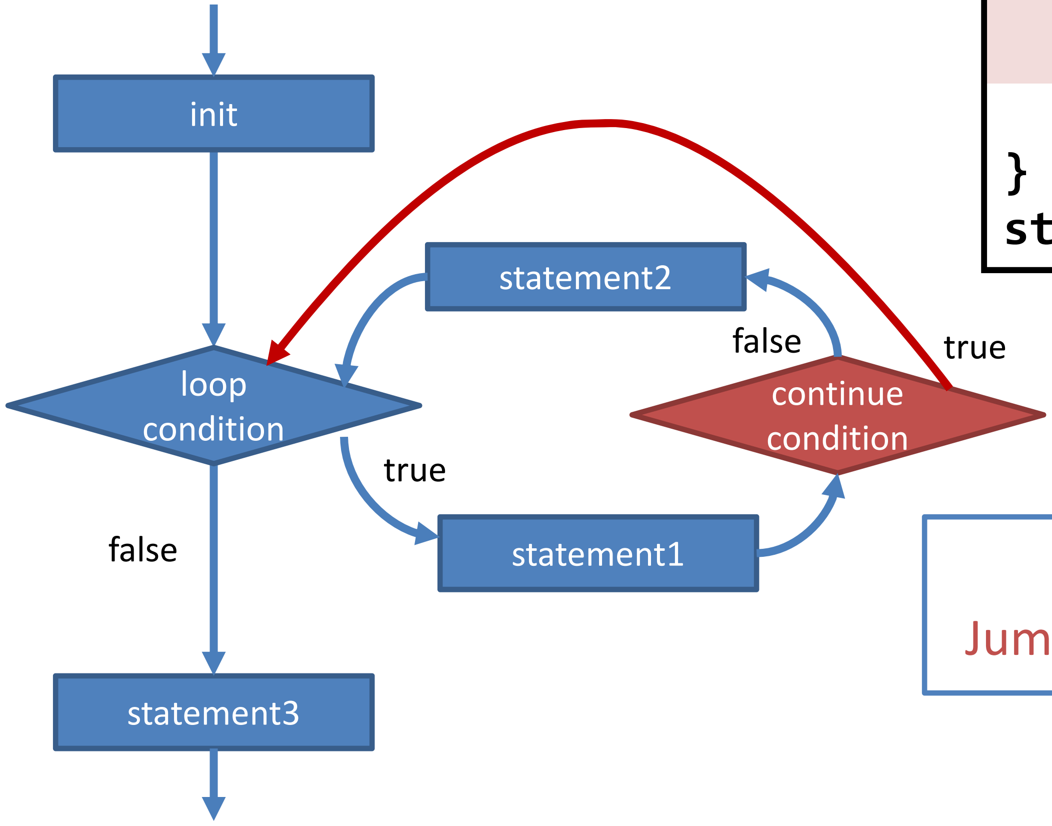
The for-loop version

Jump to "**update**"!!!

Continuation in a Loop

When executed, the **continue** statement causes the program to skip the remaining statements in the **closest enclosing loop** for the current iteration, and **jump to "update" (for loop) OR "loop condition test" (while loop)**

What is continue?



```
while ( loopCondition ) {
    statement1 ;
    if ( continueCondition )
        continue ;
    statement2 ;
}
statement3 ;
```

The while-loop version

Jump to "**loop condition test**"!!!

Examples of `continue` statement

- for-loop version

```c
int i ;

for ( i = 0 ; i < 10 ; i++ )
{
    if ( i == 3 )
        continue ;

    printf( "%d\n" , i );
}

printf( "Bye!\n" );
```

```
0
1
2
4
5
6
7
8
9
Bye!
```

What is missing?

A common mistake with `continue`

- while-loop version

```c
int i ;

i = 0 ;
while ( i < 10 )
{
    if ( i == 3 ) {
        continue ;
    }
    printf( "%d\n" , i );
    i++ ;
}
printf( "Bye!\n" );

```

Any debug in the program?

Don’t forget to “Update”!!!

- while-loop version

```
1   int i ;
2
3   i = 0 ;
4   while ( i < 10 )
5   {
6       if ( i == 3 ) {
7           i++ ;
8           continue ;
9       }
10      printf( "%d\n" , i );
11      i++ ;
12  }
13  printf( "Bye!\n" );
```

The `i++` at line 10 will be skipped when `continue` is executed. Without the `i++` at line 7, the loop will iterate forever.

Common mistake: forget to update loop variable before “continue”!!!

Summary

- `while` loop
- `for` loop
- **Nested** loop
- For more examples, please refer to slides in "04. Examples (loop)".

Note:

- C also supports "do-while" loop:

```
do {
    statement ;
} while ( condition );
```

- Characteristics:
  - **<u>At least one</u>** iteration
  - Example: ask users to input a series of values and quit when the user inputs zero.

Suggestions

- After coding, dry run and check the number of iterations!!!
  - One more or one less iteration can kill the program… bug!!!
- Call your friend "printf"!!!
  - visualize the control flow & data values during the flow
- Understand how and when to stop!!!
- Use break and continue very carefully!!!
- Read (trace code & logic) and try (& work out) more examples
- Always test and verify your code

  Think and try test data that causes different consequences!!!

Practice!

Practice!!

Practice!!!

Repeated slide: Four levels of skills

#1 Understand the flow: trace and understand code

#2 Analysis: Given a problem, carefully think and design the logic of the loops

#3 Apply: Transform the logic appropriately into for/while (with “good programming style”)

#4 Test: think about all possible consequences and evaluate your code accordingly

Practice!

Practice!!

Practice!!!