A while Loop That Counts #1

#include <stdio.h>
#include <stdlib.h>
int main ()
{ /* main */
    const int initial_sum = 0;
    const int increment = 1;
    const int program_success_code = 0;
    const int program_failure_code = -1;
    int initial_value, final_value;
    int count;
    int sum;

A while Loop That Counts #2

printf("What value would you like to ");
printf("start counting at?\n");
scanf("%d", &initial_value);
printf("What value would you like to ");
printf("stop counting at,\n");
printf(" which must be greater than ");
printf("or equal to %d?\n", initial_value);
scanf("%d", &final_value);
if (final_value < initial_value) {
    printf("ERROR: the final value %d is less\n", final_value);
    printf(" than the initial value %d.\n", initial_value);
    exit(program_failure_code);
} /* if (final_value < initial_value) */
A while Loop That Counts #3

```c
sum   = initial_sum;
count = initial_value;
while (count <= final_value) {
    sum = sum + count;
    count = count + increment;
} /* while (count <= final_value) */
printf("The sum of the integers from");
printf(" %d through %d is %d.\n", initial_value, final_value, sum);
return program_success_code;
} /* main */
```
A while Loop That Counts #4

```bash
% gcc -o whilecount whilecount.c
% whilecount
What value would you like to start counting at?
1
What value would you like to stop counting at, which must be greater than or equal to 1?
0
ERROR: the final value 0 is less than the initial value 1.
% whilecount
What value would you like to start counting at?
1
What value would you like to stop counting at, which must be greater than or equal to 1?
5
The sum of the integers from 1 through 5 is 15.
```
Count-Controlled Loops #1

On the previous slide, we saw a case of a loop that:

- executes a specific number of iterations,
- by using a counter variable,
- which is initialized to a particular initial value
- and is incremented (increased by 1) at the end of each iteration of the loop,
- until it goes beyond a particular final value:

```c
sum   = initial_sum;
count = initial_value;
while (count <= final_value) {
    sum = sum + count;
    count = count + increment;
} /* while (count <= final_value) */
```
Count-Controlled Loops #2

sum = initial_sum;
count = initial_value;
while (count <= final_value) {
    sum = sum + count;
count = count + increment;
} /* while (count <= final_value) */

We call this kind of loop a count-controlled loop.

If we express a count-controlled loop as a while loop, then the general form is:

counter = initial_value;
while (counter <= final value) {
    statement1;
    statement2;
    ...  
    counter = counter + 1;
} /* while (counter <= final value) */
counter = initial_value;
while (counter <= final value) {
    statement1;
    statement2;
    ...
    counter = counter + 1;
} /* while (counter <= final value) */
statement_after;
Arithmetic Assignment Operators #1

Some while back, we saw the following:

\[ x = x + y; \]

We learned that this statement increases the value of \( x \) by \( y \).

That is, the statement takes the old value of \( x \), adds \( y \) to it, then assigns the result of this addition to \( x \).

This kind of statement is so common that the C language has a special operator for it, called the \textit{addition assignment operator}:

\[ x += y; \]

Note that the two statements above \textit{behave identically}. 
C also has arithmetic assignment operators for the other arithmetic operations:

<table>
<thead>
<tr>
<th>This:</th>
<th>Is identical to this:</th>
<th>Operation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x += y;</code></td>
<td><code>x = x + y;</code></td>
<td>Addition assignment</td>
</tr>
<tr>
<td><code>x -= y;</code></td>
<td><code>x = x - y;</code></td>
<td>Subtraction assignment</td>
</tr>
<tr>
<td><code>x *= y;</code></td>
<td><code>x = x * y;</code></td>
<td>Multiplication assignment</td>
</tr>
<tr>
<td><code>x /= y;</code></td>
<td><code>x = x / y;</code></td>
<td>Division assignment</td>
</tr>
<tr>
<td><code>x %= y;</code></td>
<td><code>x = x % y;</code></td>
<td>Remainder assignment (int operands only)</td>
</tr>
</tbody>
</table>
Jargon: Syntactic Sugar

**Syntactic sugar** is a programming language construct that doesn’t add any new capability to the language, but makes the language a bit easier to use. Arithmetic assignment operations are syntactic sugar.
One of the most common addition assignments is:

\[ x = x + 1; \]

We learned that this statement increases the value of \( x \) by 1. That is, the statement takes the old value of \( x \), adds 1 to it, then assigns the resulting sum to \( x \).

For this statement, we could use the addition assignment operator:

\[ x += 1; \]
x = x + 1 : Programmers vs Mathematicians

https://img-9gag-fun.9cache.com/photo/a07QQ9d_700wp.webp

for Loop Lesson 1
CS1313 Fall 2022
x = x + 1;

For this statement, we could use the addition assignment operator:

x += 1;

But this statement is MUCH more common than

x += y;

for generic y, so the C language has another special operator, called the increment operator:

x++; (This is also known as the autoincrement operator.)
Increment & Decrement Operators #3

\[
x = x + 1; \\
x += 1;
\]

**Increment operator:**

\[
x++;
\]

Also:

\[
x = x - 1; \\
x -= 1;
\]

\[
x--;
\]

This is known as the **decrement operator** (and also as the **autodecrement operator**).
**Increment & Decrement Operators #4**

<table>
<thead>
<tr>
<th>This:</th>
<th>is identical to this:</th>
<th>is identical to this:</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>x++;</td>
<td>x += 1;</td>
<td>x = x + 1;</td>
<td>Increment</td>
</tr>
<tr>
<td>x--;</td>
<td>x -= 1;</td>
<td>x = x - 1;</td>
<td>Decrement</td>
</tr>
</tbody>
</table>

Note that the increment and decrement operators are syntactic sugar, just like the arithmetic assignment operators.
for Loop

A **for loop** has this form:

```plaintext
for (counter = initial_value;
     counter <= final_value; counter++) {
    statement1;
    statement2;
    ...
} /* for counter */
```
for Loop vs while Loop

A **for loop** has this form:

```c
for (counter = initial_value;
    counter <= final_value; counter++) {
    statement1;
    statement2;
    ...
} /* for counter */
```

A **for loop** behaves **exactly the same** as a count-controlled **while loop**:

```c
counter = initial_value;
while (counter <= final_value) {
    statement1;
    statement2;
    ...
    counter = counter + 1;
} /* while (counter <= final_value) */
```
for Loop Flowchart

```
for (counter = initial_value;
     counter <= final_value;
     counter++) {
    statement1;
    statement2;
    ...
} /* for counter */
statement_after;
```

Notice that the for loop flowchart is identical to the while loop flowchart on slide 8.
#include <stdio.h>

int main ()
{
    int count;
    int sum;

    sum = 0;
    count = 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;

    printf("count = %d\n", count);
    printf("sum   = %d\n", sum);
    return 0;
}
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum;

    sum = 0;
    count = 1;
    while (count <= 5) {
        sum = sum + count;
        count += 1;
    } /* while (count <= 5) */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
Three Programs That Behave the Same #3

```c
#include <stdio.h>
int main ()
{ /* main */
    int count;
    int sum;

    sum = 0;
    for (count = 1; count <= 5; count++) {
        sum = sum + count;
    } /* for count */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
```
Three Programs That Behave the Same #4

```bash
% gcc -o manycountstmts manycountstmts.c
% manycountstmts
count = 6
sum   = 15

% gcc -o while_loop while_loop.c
% while_loop
count = 6
sum   = 15

% gcc -o for_loop for_loop.c
% for_loop
count = 6
sum   = 15
```
for Loop Example

% cat product_loop.c
#include <stdio.h>
int main ()
{ /* main */
    int count;
    int product;

    product = 1;
    for (count = 1; count <= 5; count++) {
        product = product * count;
    } /* for count */
    printf("After the loop: count = %d, ", count);
    printf("product = %d\n", product);
    return 0;
} /* main */

% gcc -o product_loop product_loop.c
% product_loop
After the loop: count = 6, product = 120
for Loop Behavior #1

for (count = 1; count <= 5; count++) {
    product = product * count;
} /* for count */

1. The **loop initialization** is performed; typically, the **loop control variable** (also known as the **loop counter** or the **loop index**) is assigned an **initial value** (also known as the **lower bound**).

**NOTE**: The loop initialization is performed only the **FIRST TIME** that the for statement is reached.

Once a loop is underway, that loop’s initialization **DOESN’T** get executed again.

We refer to each trip through the body of the loop as an **iteration**.
for Loop Behavior #2

```java
for (count = 1; count <= 5; count++) {
    product = product * count;
} /* for count */
```

2. The loop **continuation condition** is evaluated, and if the loop continuation condition evaluates to false (0), then the `for` loop body is skipped, and the program continues on from the first statement after the `for` loop’s block close. But, if the loop continuation condition evaluates to true (1), then enter the loop body.

We refer to each trip through the body of the loop as an **iteration**.
for Loop Behavior #3

```java
for (count = 1; count <= 5; count++) {
    product = product * count;
} /* for count */
```

3. Each statement inside the **loop body** is executed in sequence.

We refer to each trip through the body of the loop as an **iteration**.
for Loop Behavior #4

```java
for (count = 1; count <= 5; count++) {
    product = product * count;
} /* for count */
```

4. When the end of the loop body is reached (indicated by the block close associated with the block open of the `for` statement), the loop counter is changed by the *loop change statement*, typically (though not always) by incrementing.

We refer to each trip through the body of the loop as an *iteration*. 
for Loop Behavior #5

```c
for (count = 1; count <= 5; count++) {
    product = product * count;
} /* for count */
```

5. REPEAT from step 2.

(Step 1, the loop initialization, gets executed only the FIRST TIME that the for statement is reached.)

We refer to each trip through the body of the loop as an iteration.
The above program fragment behaves identically the same as:

```c
int product = 1;  /* product = 1 */
int count;        /* count is undefined */
count = 1;        /* count == 1, product == 1 */
product *= count; /* count == 1, product == 1 */
count++;          /* count == 2, product == 1 */
product *= count; /* count == 2, product == 2 */
count++;          /* count == 3, product == 3 */
product *= count; /* count == 3, product == 6 */
count++;          /* count == 4, product == 6 */
product *= count; /* count == 4, product == 24 */
count++;          /* count == 5, product == 24 */
product *= count; /* count == 5, product == 120 */
count++;          /* count == 6, product == 120 */
```
Why Have \texttt{for} Loops?

If a count-controlled loop can be expressed as a \texttt{while} loop, then why have \texttt{for} loops at all?

Imagine that a count-controlled loop has a very long loop body, for example longer than a screenful of source code text.

In that case, the change statement (for example, incrementing the loop counter variable) could be very far away from the initialization and the condition.

In which case, looking at the \texttt{while} statement, you couldn’t immediately understand its count-controlled behavior.

But by putting all of the count-control code in a single \texttt{for} statement, you can look at just the \texttt{for} statement and immediately understand the count-control behavior.