



Chapter Four: Loops

Chapter Goals

- To implement **while**, **for** and **do** loops
- To avoid infinite loops and off-by-one errors
- To understand nested loops
- To implement programs that read and process data sets
- To use a computer for simulations

What Is the Purpose of a Loop?

A loop is a statement that is used to:

execute one or more statements
repeatedly until a goal is reached.

Sometimes these one-or-more statements
will not be executed at all
—if that's the way to reach the goal

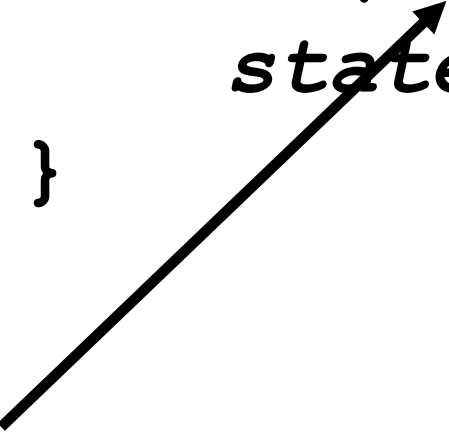
The Three Loops in C

C has these three looping statements:

while
for
do while

The while Loop

```
while (condition) {  
    statements  
}
```



The *condition* is some kind of test
(the same as it was in the `if` statement)

The while Loop

```
while (condition) {  
    statements  
}
```

The statements are repeatedly executed
until the condition is **false**

Using a Loop to Solve an Investment Problem.

Starting with \$10,000, how many years until we have at least \$20,000?

The algorithm:

1. Start with a year value of 0 and a balance of \$10,000.
2. **Repeat** the following steps
while the balance is less than \$20,000:
 - Add 1 to the year value.
 - Compute the interest by multiplying the balance value by 0.05 (5 percent interest) (will be a `const`, of course).
 - Add the interest to the balance.
3. Report the final year value as the answer.

Using a Loop to Solve an Investment Problem.

2. **Repeat** the following steps
while the balance is less than \$20,000:

“Repeat .. while” in the problem indicates a loop is needed. To reach the goal of being able to report the final year value, adding and multiplying must be repeated some unknown number of times.

Using a Loop to Solve the Investment Problem.

The statements to be controlled are:

- Incrementing the **year** variable
- Computing the **interest** variable, using a **const** for the **RATE**
- Updating the **balance** variable by adding the **interest**

```
year++;
```

```
double interest = balance * RATE / 100;
```

```
balance = balance + interest;
```

Using a Loop to Solve the Investment Problem.

The condition, which indicates when to ***stop*** executing the statements, is this test:

(balance < TARGET)

Using a Loop to Solve the Investment Problem.

Here is the complete `while` statement:

```
while (balance < TARGET) {  
    year++;  
    double interest = balance * RATE / 100;  
    balance = balance + interest;  
}
```

Using a Loop to Solve the Investment Problem.

Notice that `interest` is defined *inside* the loop and that `year` and `balance` had to have been defined *outside* the loop.

```
while (balance < TARGET) {  
    year++;  
    double interest = balance * RATE / 100;  
    balance = balance + interest;  
}
```

A new `interest` variable to be created in *each* iteration.

`year` and `balance` are used for *all* iterations.

The while Statement

This variable is defined outside the loop and updated in the loop.

If the condition never becomes false, an infinite loop occurs.

```
double balance = 0;
.  
.  
.  
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Beware of "off-by-one" errors in the loop condition.

Don't put a semicolon here!

This variable is created in each loop iteration.

These statements are executed while the condition is true.

Lining up braces is a good idea.

Braces are not required if the body contains a single statement, but it's good to always use them.

The Complete Investment Program

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    const double RATE = 5;
    const double INITIAL_BALANCE = 10000;
    const double TARGET = 2 * INITIAL_BALANCE;

    double balance = INITIAL_BALANCE;
    int year = 0;
    while (balance < TARGET) {
        year++;
        double interest = balance * RATE / 100;
        balance = balance + interest;
    }

    printf("The investment doubled after %d years.\n", year);
    return EXIT_SUCCESS;
}
```

Program Run

Check the loop condition

balance = 10000

year = 0

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

The condition is true

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Execute the statements in the loop

balance = 10000

year = 0

interest = ?

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```


Program Run

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

The condition is true

Execute the statements in the loop

balance = 10000

year = 1

interest = ?

```
while (balance < TARGET)
```

```
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

The condition is true

Execute the statements in the loop

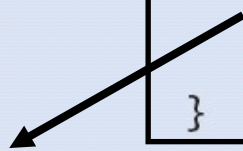
balance = 10000

year = 1

interest = 500

```
while (balance < TARGET)
```

```
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```



Program Run

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

The condition is true

Execute the statements in the loop

balance = 10500

year = 1

interest = 500

```
while (balance < TARGET)
```

```
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

The condition is true

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Execute the statements in the loop

balance = 10500

year = 1

interest = 500

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

Check the loop condition

balance = 10500

year = 1

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

The condition is still true

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Execute the statements in the loop

balance = 10500

year = 1

interest = ?

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```


Program Run

The condition is still true

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```


Execute the statements in the loop

balance = 10500

year = 2

interest = ?

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```



Program Run

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

The condition is still true

Execute the statements in the loop

balance = 10500

year = 2

interest = 525

```
while (balance < TARGET)
```

```
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```


Program Run

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

The condition is still true

Execute the statements in the loop

balance = 11025

year = 2

interest = 525

```
while (balance < TARGET)
```

```
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

The condition is still true

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Execute the statements in the loop

balance = 11025

year = 2

interest = 525

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

Check the loop condition

balance = 11025

year = 2

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

before entering while's body		at the end of while's body		
balance	year	interest	balance	year
10000.00	0	500.00	10500.00	1
10500.00	1	525.00	11025.00	2

...this process goes on
for 15 iterations...

Program Run

...this process goes on
for 15 iterations...

before entering while's body		at the end of while's body		
balance	year	interest	balance	year
10000.00	0	500.00	10500.00	1
10500.00	1	525.00	11025.00	2
11025.00	2	551.25	11576.25	3

Program Run

...this process goes on
for 15 iterations...

before entering while's body		at the end of while's body		
balance	year	interest	balance	year
10000.00	0	500.00	10500.00	1
10500.00	1	525.00	11025.00	2
11025.00	2	551.25	11576.25	3
11576.25	3	578.81	12155.06	4
12155.06	4	607.75	12762.82	5

Program Run

...this process goes on
for 15 iterations...

before entering while's body		at the end of while's body		
balance	year	interest	balance	year
10000.00	0	500.00	10500.00	1
10500.00	1	525.00	11025.00	2
11025.00	2	551.25	11576.25	3
11576.25	3	578.81	12155.06	4
12155.06	4	607.75	12762.82	5
12762.82	5	638.14	13400.96	6
13400.96	6	670.05	14071.00	7
14071.00	7	703.55	14774.55	8
14774.55	8	738.73	15513.28	9

Program Run

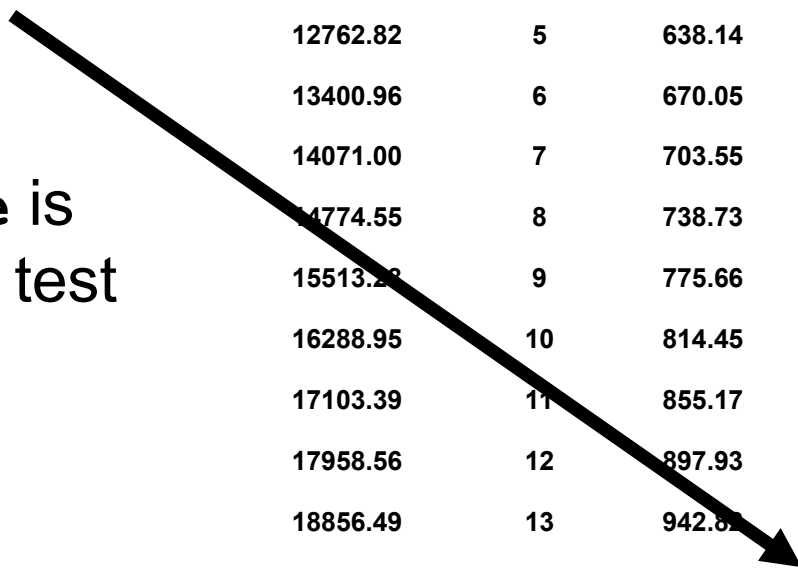
...this process goes on
for 15 iterations...

before entering while's body		at the end of while's body		
balance	year	interest	balance	year
10000.00	0	500.00	10500.00	1
10500.00	1	525.00	11025.00	2
11025.00	2	551.25	11576.25	3
11576.25	3	578.81	12155.06	4
12155.06	4	607.75	12762.82	5
12762.82	5	638.14	13400.96	6
13400.96	6	670.05	14071.00	7
14071.00	7	703.55	14774.55	8
14774.55	8	738.73	15513.28	9
15513.28	9	775.66	16288.95	10
16288.95	10	814.45	17103.39	11
17103.39	11	855.17	17958.56	12
17958.56	12	897.93	18856.49	13
18856.49	13	942.82	19799.32	14

Program Run

...this process goes on
for 15 iterations...

...until the **balance** is
over \$20,000 and the test
becomes **false**.



before entering while's body		at the end of while's body		
balance	year	interest	balance	year
10000.00	0	500.00	10500.00	1
10500.00	1	525.00	11025.00	2
11025.00	2	551.25	11576.25	3
11576.25	3	578.81	12155.06	4
12155.06	4	607.75	12762.82	5
12762.82	5	638.14	13400.96	6
13400.96	6	670.05	14071.00	7
14071.00	7	703.55	14774.55	8
14774.55	8	738.73	15513.28	9
15513.28	9	775.66	16288.95	10
16288.95	10	814.45	17103.39	11
17103.39	11	855.17	17958.56	12
17958.56	12	897.93	18856.49	13
18856.49	13	942.81	19799.32	14
19799.32	14	989.97	20789.28	15
while statement is over				

Program Run

After 15 iterations

balance = 20789.28

year = 15

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Program Run

balance = 20789.28

year = 15

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

The condition is
no longer true

Program Run

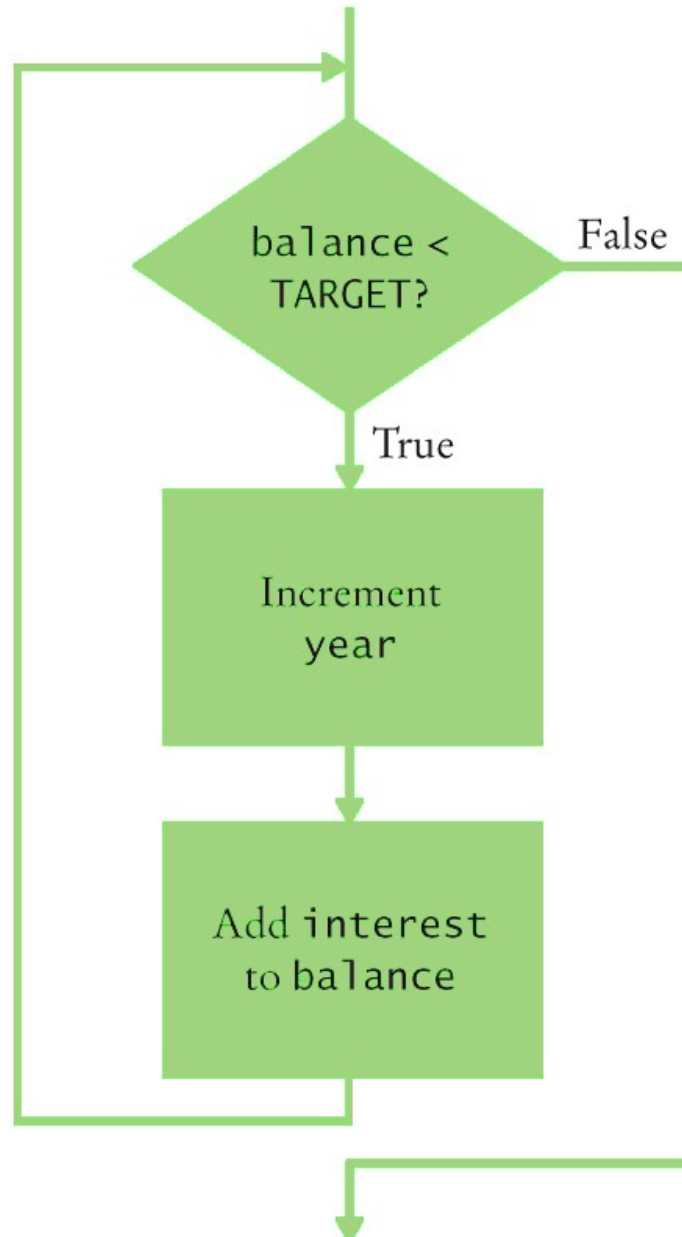
Execute the statement following the loop

balance = 20789.28

year = 15

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
printf("%d ", year);
```

Flowchart of the Investment Calculation's while Loop

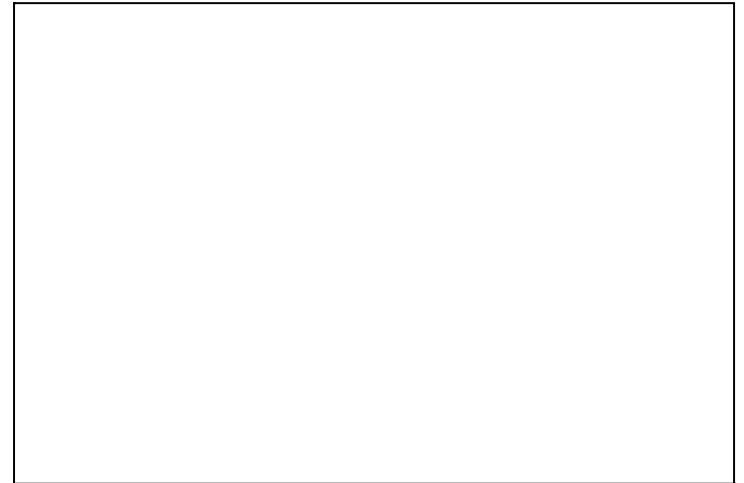


Example of Normal Execution

while loop to hand-trace

```
i = 5;
while (i > 0) {
    printf("%d ", i);
    i--;
}
```

What is the output?



When `i` is 0, the Loop Condition is `false`, and the Loop Ends

`while` loop

```
i = 5;
while (i > 0) {
    printf("%d ", i);
    i--;
}
```

The output

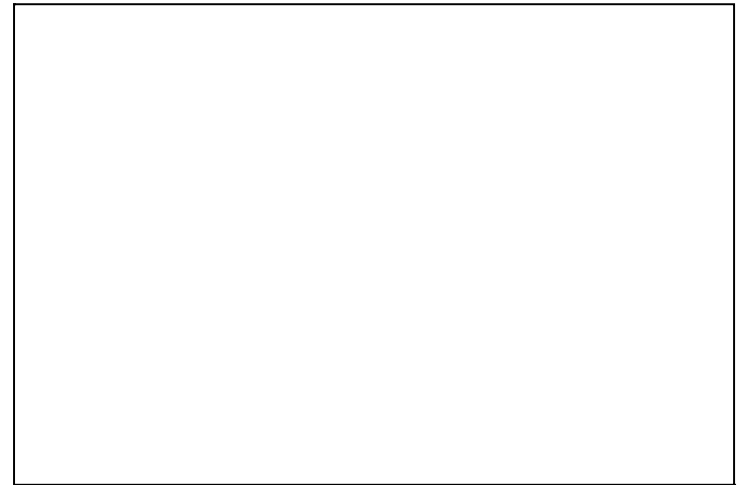
```
5 4 3 2 1
```

Example of a Problem – An Infinite Loop

while loop to hand-trace

```
i = 5;
while (i > 0) {
    printf("%d ", i);
    i++;
}
```

What is the output?



Example of a Problem – An Infinite Loop

i is set to 5

**The `i++;` statement makes `i` get bigger and bigger
the condition will never become false –
an infinite loop**

while loop

```
i = 5;  
while (i > 0) {  
    printf("%d ", i);  
    i++;  
}
```

The output never ends

5 6 7 8 9 10 11...

Another Normal Execution – No Errors

while loop to hand-trace

```
i = 5;
while (i > 5) {
    printf("%d ", i);
    i--;
}
```

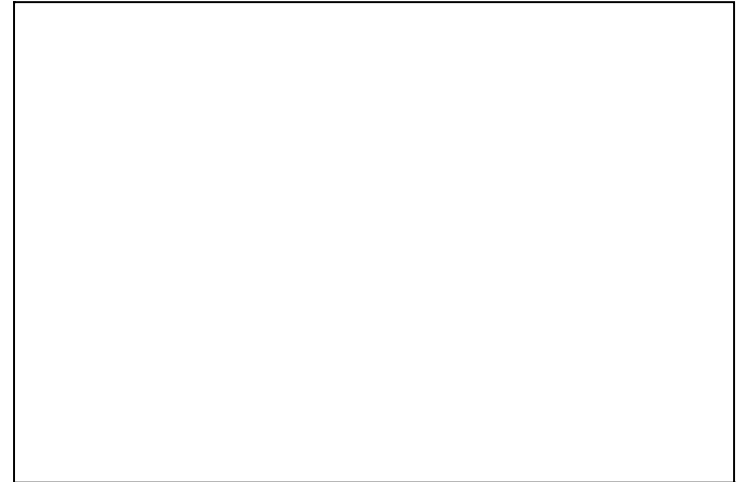
What is the output?

Another Normal Execution – No Errors

`while` loop

```
i = 5;
while (i > 5) {
    printf("%d ", i);
    i--;
}
```

There is (correctly) no output



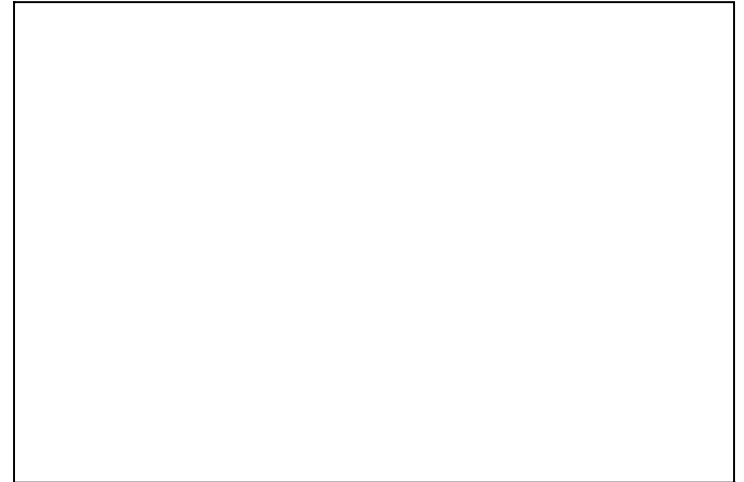
The expression `i > 5` is initially false, so the statements are never executed.

Another Normal Execution – No Errors

`while` loop

```
i = 5;
while (i > 5) {
    printf("%d ", i);
    i--;
}
```

There is (correctly) no output



This is not a error.

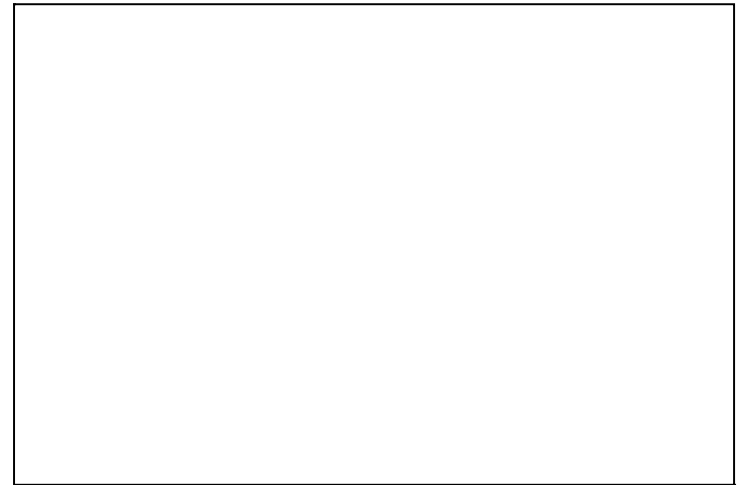
Sometimes we *do not* want to execute the statements
unless the test is true.

Normal Execution with Another “Programmer’s Error”

while loop to hand-trace

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

What is the output?



Normal Execution with Another “Programmer’s Error”

The programmer probably thought:
“Stop when `i` is less than 0”.

However, the loop condition controls
when the loop is *executed* - not when it *ends*.

`while` loop

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

Again, there is no output

A Very Difficult Error to Find (especially after looking for it for hours and hours!)

`while` loop to hand-trace

```
i = 5;
while (i > 0);
{
    printf("%d ", i);
    i--;
}
```

What is the output?

A Very Difficult Error to Find (especially after looking for it for hours and hours!)

Another infinite loop – caused by a single character:

That semicolon causes the **while** loop to have an “empty body” which is executed forever.

The **i** in (**i** > 0) is never changed.

while loop


```
i = 5;
while (i > 0);
{
    printf("%d ", i);
    i--;
}
```

There is no output!

Common Error – Infinite Loops

- Forgetting to update the variable used in the condition is common.
- In the investment program, it might look like this.

```
year = 1;
while (year <= 20) {
    balance = balance * (1 + RATE / 100);
}
```




- The variable **year** is not updated in the body

Common Error – Are We There Yet?

When doing something repetitive,
most of us want to know when we are done.

For example, you may think,
“I want to get at least \$20,000,”
and set the loop condition to

```
while (balance >= TARGET)
```



wrong test

Common Error – Are We There Yet?

But the **while** loop thinks the opposite:
How long am I allowed to keep going?

What is the correct loop condition?

```
while ( )
```

Common Error – Are We There Yet?

But the **while** loop thinks the opposite:
How long am I allowed to keep going?

What is the correct loop condition?

```
while (balance < TARGET)
```

In other words:

“Keep at it while the balance
is less than the target”.

Common Error – Are We There Yet?

When writing a loop condition, don't ask, "Are we there yet?"

The *condition* determines how long the loop will keep going.

Common Error – Off-by-One Errors

In the code to find when we have doubled our investment:

Do we start the variable for the years
at 0 or 1 years?

Do we test for **< TARGET**
or for **<= TARGET**?

Common Error – Off-by-One Errors

- Maybe if you start trying some numbers and add +1 or -1 until you get the right answer you can figure these things out.
- It will most likely take a very long time to try ALL the possibilities.
- No, just try a couple of “test cases” (**while *thinking***).

Use Thinking to Decide!

- Consider starting with \$100 and a **RATE** of 50%.
 - We want \$200 (or more).
 - At the end of the first year,
the balance is \$150 – not done yet
 - At the end of the second year,
the balance is \$225 – definitely over **TARGET**
and we are done.
- We made two increments.

What must the original value be so that we end up with 2?

Zero, of course.

Use Thinking to Decide!

Another way to think about the initial value is:

Before we even enter the loop, what is the correct value?

Most often it's zero.

< vs. <= (More Thinking)

- Figure out what you want:

“we want to keep going until
we have doubled the balance”

- So you might have used:

(balance < TARGET)

< vs. <= (More Thinking)

- But consider, did you really mean:

“...to have *at least* doubled...”

Exactly twice as much would happen with
a **RATE** of 100% - the loop should stop then

- So the test must be **(balance <= TARGET)**

Problem Solving: Hand-Tracing

Hand-tracing is a method of checking your work.

To do a hand-trace, write your variables on a sheet of paper and mentally execute each step of your code...

writing down the values of the variables
as they are changed in the code.

Cross out the old value and write down the new value as they are changed – that way you can also see the history of the values.

Problem Solving: Hand-Tracing

To keep up with which statement is about to be executed you should use a marker.

Preferably something that doesn't obliterate the code:



Like a paper clip.

(No, not that infamous one!)

Problem Solving: Hand-Tracing

Consider this example. What value is displayed?

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```


```
int n = 1729;
int sum = 0;
while (n > 0) {
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
printf("%d", sum);
```

There are three variables: `n`, `sum`, and `digit`.

n	sum	digit

Problem Solving: Hand-Tracing

The first two variables are initialized with 1729 and 0 before the loop is entered.




```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```

n	sum	digit
1729	0	

Problem Solving: Hand-Tracing

Because n is greater than zero, enter the loop. The variable `digit` is set to 9 (the remainder of dividing 1729 by 10). The variable `sum` is set to $0 + 9 = 9$.




```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```

n	sum	digit
1729	0	
	9	9

Problem Solving: Hand-Tracing

Finally, `n` becomes 172. (Recall that the remainder in the division $1729 / 10$ is discarded because both arguments are integers.)

Cross out the old values and write the new ones under the old ones.



```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```

n	sum	digit
1729	0	
172	9	9

Problem Solving: Hand-Tracing

Now check the loop condition again.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```



Because n is still greater than zero, repeat the loop. Now $digit$ becomes 2, sum is set to $9 + 2 = 11$, and n is set to 17.

n	sum	digit
1729	0	
172	9	9
17	11	2

Problem Solving: Hand-Tracing

Repeat the loop once again, setting digit to 7, sum to $11 + 7 = 18$, and n to 1.


n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7

Problem Solving: Hand-Tracing

Enter the loop for one last time. Now digit is set to 1, sum to 19, and n becomes zero.

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7
0	19	1

Problem Solving: Hand-Tracing



```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```

**Because n equals zero,
this condition is not true.**

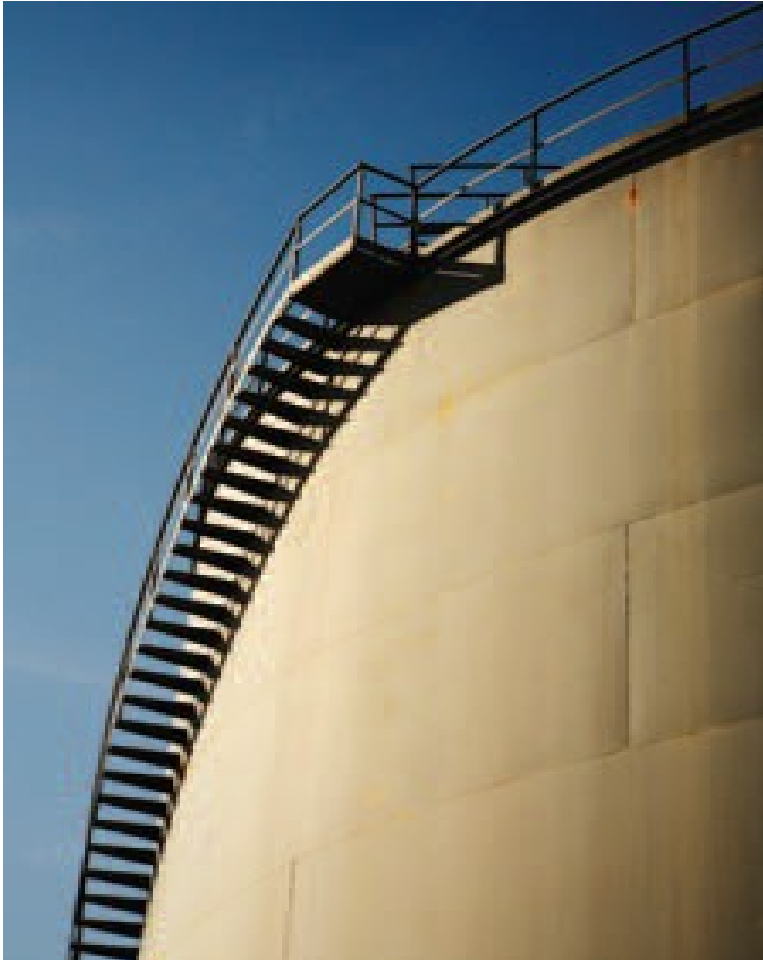
Problem Solving: Hand-Tracing

The condition $n > 0$ is now false. Continue with the statement after the loop.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
cout << sum << endl;
```

n	sum	digit	output
1729	0		
172	9	9	
17	11	2	
1	18	1	
0	19	1	19

The for Loop



To execute statements a
certain number of times

The for Loop

Often you will need to execute a sequence of statements a given number of times.

You could use a **while** loop for this.

```
counter = 1; // Initialize the counter
while (counter <= 10) { // Check the counter
    printf("%d ", counter);
    counter++; // Update the counter
}
```


The for Loop Is Better than while for Doing Certain Things

Consider this code which writes the values
1 through 10 on the screen:

```
int count = 1; // Initialize the counter  
while (count <= 10) // Check the counter  
{  
    printf("%d ", count);  
    count++; // Update the counter  
}
```

The diagram illustrates the components of a while loop with arrows pointing from labels to the corresponding code lines:

- initialization points to int count = 1;
- condition points to while (count <= 10)
- statements points to printf("%d ", count);
- update points to count++;

The for Loop

C has a statement custom made for this sort of processing:

```
for (counter = 1; counter <= 10; counter++) {  
    printf("%d ", counter);  
}
```

The for Loop Is Better than while for Doing Certain Things

Doing something a certain number of times or causing a variable to take on a sequence of values is so common, C has a statement just for that:


```
int count;  
for (count = 1; count <= 10; count++)  
{  
    printf("%d ", count);  
}
```

The diagram illustrates the components of a C for loop. Four labels at the bottom are connected by arrows to specific parts of the for loop syntax: 'initialization' points to 'count = 1', 'condition' points to 'count <= 10', 'statements' points to 'printf("%d ", count);', and 'update' points to 'count++'.

initialization *condition* *statements* *update*

The for Loop

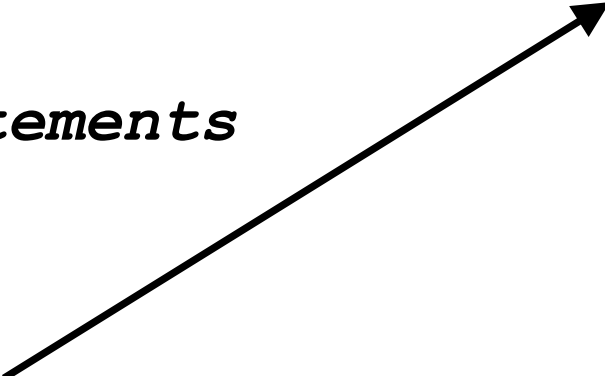
```
for (initialization; condition; update)  
{  
    statements  
}
```



The *initialization* is code that happens once, before the check is made, in order to set up for counting how many times the *statements* will happen. The loop variable is created here.

The for Loop

```
for (initialization; condition; update)  
{  
    statements  
}
```



The *condition* is code that tests to see if the loop is done. When this test is false, the **for** statement is over and we go on to the next statement.

The for Loop

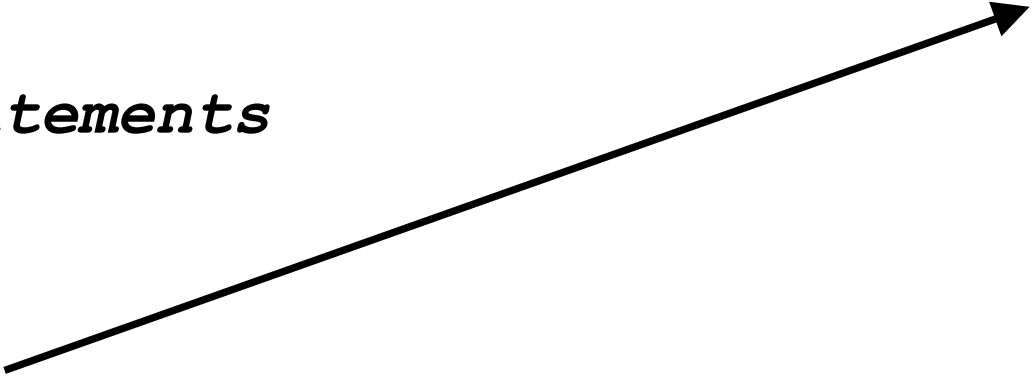
```
for (initialization; condition; update)  
{  
    statements  
}
```



The *statements* are repeatedly executed
- until the condition is false.

The for Loop

```
for (initialization; condition; update)  
{  
    statements  
}
```



The *update* is code that causes the condition to eventually become false.

Usually it's incrementing or decrementing the loop variable.

The `for` Loop

Some people call the `for` loop *count-controlled*.

In contrast, the `while` can be called an *event-controlled* loop because it executes until an event occurs (for example, when the balance reaches the target).

The for Loop

Another commonly-used term for a count-controlled loop is *definite*.

You know from the outset that the loop body will be executed a definite number of times—ten times in our example.

In contrast, you did not know how many iterations it would take to accumulate a target balance in the **while** loop code.

Such a loop is called *indefinite*.

Execution of a for Statement

Consider this `for` statement:

```
int counter;  
for (counter = 1; counter <= 10; counter++) {  
    printf("%d\n", counter);  
}
```

1 Initialize counter

counter =

```
for (counter = 1; counter <= 10; counter++)  
{  
    cout << counter << endl;  
}
```

2 Check counter

counter =

```
for (counter = 1; counter <= 10; counter++)  
{  
    cout << counter << endl;  
}
```

3 Execute loop body

counter =

```
for (counter = 1; counter <= 10; counter++)  
{  
    cout << counter << endl;  
}
```

4 Update counter

counter =

```
for (counter = 1; counter <= 10; counter++)  
{  
    cout << counter << endl;  
}
```

5 Check counter again

counter =

```
for (counter = 1; counter <= 10; counter++)  
{  
    cout << counter << endl;  
}
```

The for Statement

SYNTAX 4.2 for Statement

This *initialization* happens once before the loop starts.

The loop is executed while this *condition* is true.

This *update* is executed after each iteration.

These three expressions should be related.

The variable `i` is defined only in this `for` loop.

```
for (int i = 5; i <= 10; i++)  
{  
    sum = sum + i;  
}
```

This loop executes 6 times.

Scope of the Loop Variable – Part of the `for` or Not?

- The “loop variable” when defined as part of the `for` statement cannot be used before or after the `for` statement – it only exists as part of the `for` statement and should not need to be used anywhere else in a program.
- A `for` statement can use variables that are not part of it, but they should not be used as the loop variable.

(In an earlier example, `counter` was defined before the loop – so it does work. Normally `counter` would be defined in the *initialization*.)

The `for` Can Count Up or Down

A `for` loop can count down instead of up:

```
for (int counter = 10; counter >= 0; counter--)...
```

The increment or decrement need not be in steps of 1:

```
for (int cntr = 0; cntr <= 10; cntr += 2)...
```

Notice that in these examples, the loop variable is defined **in** the *initialization* (where it really should be!).

Solving a Problem with a `for` Statement

- Earlier we determined the number of years it would take to (at least) double our balance.
- Now let's see the interest in action:
 - We want to print the balance of our savings account over a five-year period.

The "...over a five-year period" indicates that a `for` loop should be used.

Because we know how many times the statements must be executed, we choose a `for` loop.

Solving a Problem with a for Statement

The output should look something like this:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

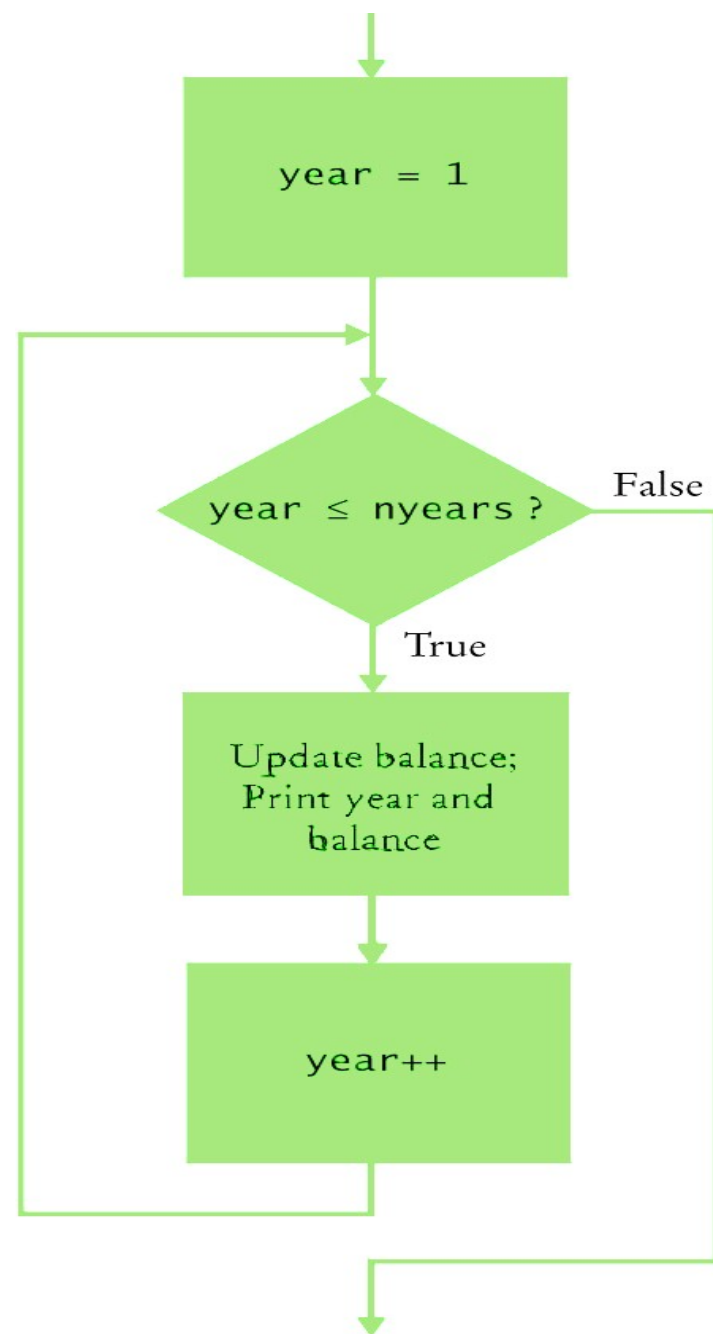
Solving a Problem with a `for` Statement

The pseudocode:

```
for (int year = 1; year <= nyears; year++) {  
    Update balance.  
    Print year and balance.  
}
```

The for Loop

Flowchart of
the investment
calculation
using
a **for** loop



Solving a Problem with a `for` Statement

Two statements should happen five times.

So use a `for` statement.

They are:

 update balance

 print year and balance

```
for (int year = 1; year <= nyears; year++) {  
    // update balance  
    // print year and balance  
}
```

The Modified Investment Program Using a for Loop

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    const double RATE = 5;
    const double INITIAL_BALANCE = 10000;
    double balance = INITIAL_BALANCE;

    int nyears = 0;
    printf("Enter number of years: ");
    scanf("%d", &nyears);

    for (int year = 1; year <= nyears; year++) {
        balance = balance * (1 + RATE / 100);
        printf("%4d %-10.2f\n", year, balance);
    }

    return EXIT_SUCCESS;
}
```

The Modified Investment Program Using a for Loop

A run of the program:

```
Enter number of years: 10
```

```
1 10500.00
2 11025.00
3 11576.25
4 12155.06
5 12762.82
6 13400.96
7 14071.00
8 14774.55
9 15513.28
10 16288.95
```

Example of Normal Execution

for loop to hand-trace

```
for (int i = 0;  
    i <= 5;  
    i++) {  
    printf("%d ", i);  
}
```

What is the output?

Example of Normal Execution

for loop

```
for (int i = 0;  
    i <= 5;  
    i++) {  
    printf("%d ", i);  
}
```

The output

```
0 1 2 3 4 5
```

Note that the output statement is
executed six times, not five

Example of Normal Execution – Going in the Other Direction

for loop to hand-trace

```
for (int i = 5;  
    i >= 0;  
    i--) {  
    printf("%d ", i);  
}
```

What is the output?

Example of Normal Execution – Going in the Other Direction

Again six executions of the output statement occur.

for loop

```
for (int i = 5;  
    i >= 0;  
    i--) {  
    printf("%d ", i);  
}
```

The output

```
5 4 3 2 1 0
```

Example of Normal Execution – Taking Bigger Steps

for loop to hand-trace

```
for (int i = 0;  
    i < 9;  
    i += 2) {  
    printf("%d ", i);  
}
```

What is the output?

Example of Normal Execution – Taking Bigger Steps

for loop

```
for (int i = 0;  
    i < 9;  
    i += 2) {  
    printf("%d ", i);  
}
```

The output

0 2 4 6 8

The “step” value can be added to or subtracted from the loop variable.

Here the value 2 is added.

There are only 5 iterations, though.

Infinite Loops Can Occur in `for` Statements

The danger of using `==` and/or `!=`

`for` loop to hand-trace

```
for (int i = 0;  
     i != 9;  
     i += 2) {  
    printf("%d ", i);  
}
```

What is the output?

Infinite Loops Can Occur in `for` Statements

`==` and `!=` are best avoided
in the check of a `for` statement

`for` loop

```
for (int i = 0;  
    i != 9;  
    i += 2) {  
    printf("%d ", i);  
}
```

The output never ends

0 2 4 6 8 10 12...

Example of Normal Execution – Taking Even Bigger Steps

for loop to hand-trace

```
for (int i = 1;  
    i <= 20;  
    i *= 2) {  
    printf("%d ", i);  
}
```

What is the output?



The update can be any expression

Example of Normal Execution – Taking Even Bigger Steps

for loop

```
for (int i = 1;  
     i <= 20;  
     i *= 2) {  
    printf("%d ", i);  
}
```

The output

```
1 2 4 8 16
```

The “step” can be multiplicative or any valid expression

Confusing Everyone, Most Likely Including Yourself

- A **for** loop is an *idiom* for a loop of a particular form. A value runs from the start to the end, with a constant increment or decrement.
- As long as all the expressions in a **for** loop are valid, the compiler will not complain.

Confusing Everyone, Most Likely Including Yourself

A **for** loop should only be used to cause a loop variable to run, with a consistent increment, from the start to the end of a sequence of values.

Or you could write this (it works, but ...)

```
for (printf("Inputs: ");  
     scanf("%d", &x);  
     sum += x) {  
count++;  
}
```

Know Your Bounds – Closed vs. Half-open

- The start and end values should match the task the `for` loop is solving.
- The range $3 \leq n \leq 17$ is *closed*, both end points are included so the `for` loop is:

```
for (int n = 3; n <= 17; n++)...
```

Know Your Bounds – Closed vs. Half-open

- When dealing with arrays (in a later chapter), you'll find that if there are n items in an array, you must deal with them using the range $[0..n)$.
So the `for` loop for arrays is:

```
for (int i=0; i<n; i++ )...
```

- This still executes the statements n times.

Many coders use this *half-open* form for **every** problem involving doing something n times.

Counting Iterations

- Finding the correct lower and upper bounds and the correct check for an iteration can be confusing.
 - Should you start at 0 or at 1?
 - Should you use $\leq b$ or $< b$ as a termination condition?
- Counting the number of iterations is a very useful device for better understanding a loop.

Counting Iterations

Counting is easier for loops with *half-open* bounds.

The loop

```
for (i = a; i < b; i++)...
```

executes the statements $(b - a)$ times
and when a is 0: b times.

Counting Iterations

The loop with closed bounds,

```
for (i = a; i <= b; i++) ...
```

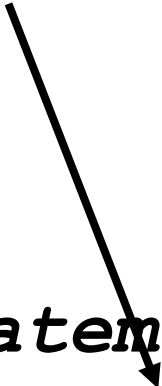
is executed $(b - a) + 1$ times.

That "+1" is the source of many programming errors.

The do Loop

The **while** loop's condition test is the first thing that occurs in its execution.

The **do** loop (or **do-while** loop) has its condition tested only after at least one execution of the statements.



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

The do Loop

This means that the `do` loop should be used only when the statements must be executed before there is any knowledge of the condition.

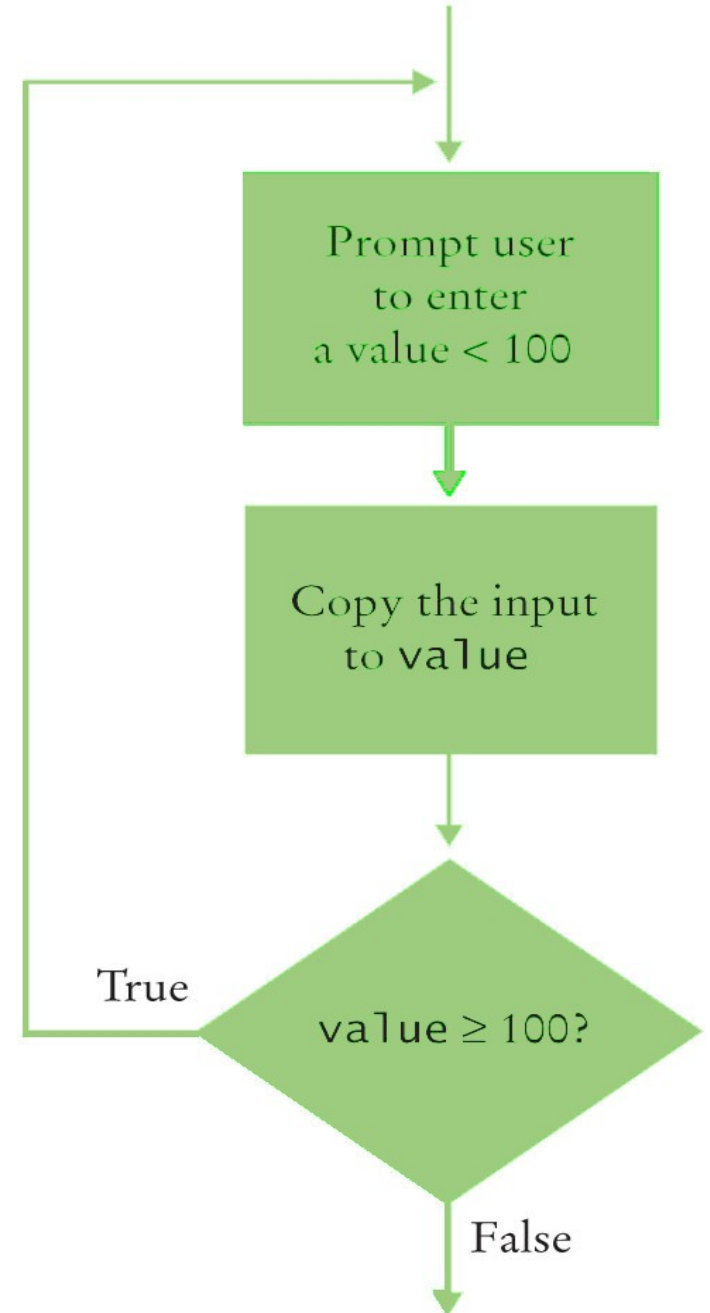
This also means that the `do` loop is the least used loop.

The do Loop

What problems require something to have happened before the testing in a loop?

Getting valid user input is often cited.

Here is the flowchart for the problem in which the user is supposed to enter a value less than 100 and processing must not continue until they do.



The do Loop

Here is the code:

```
int value = 0;
do {
    printf("Enter a value < 100: ");
    scanf("%d ", &value);
} while (value >= 100);
```

In this form, the user sees the same prompt each time until they enter valid input.

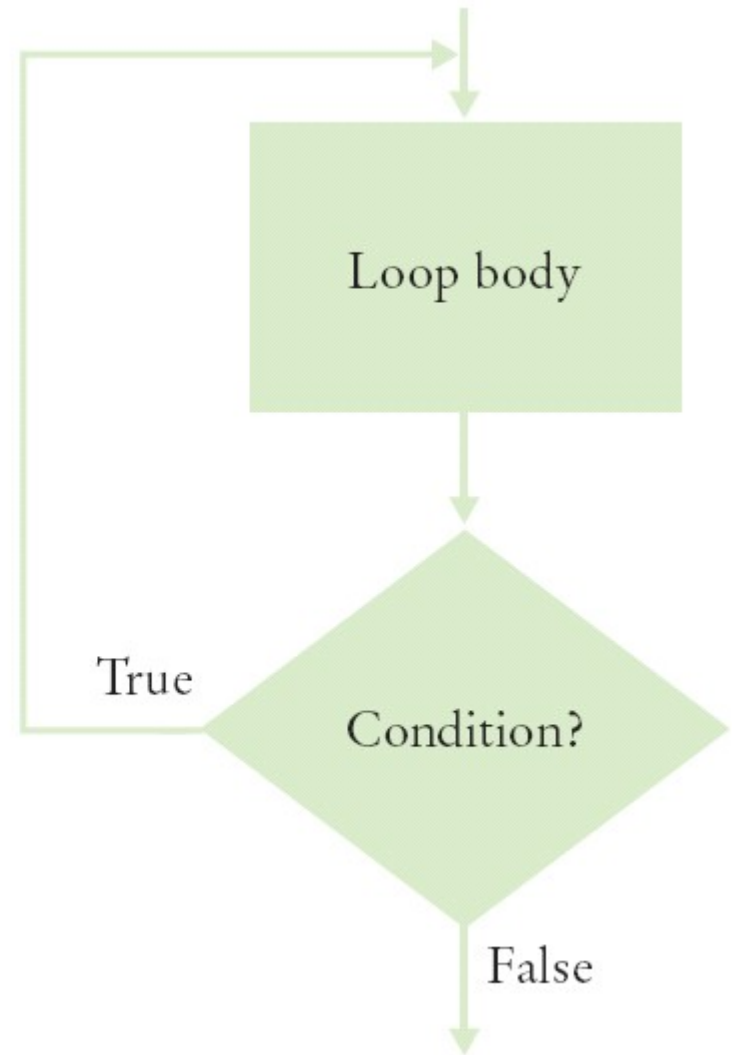
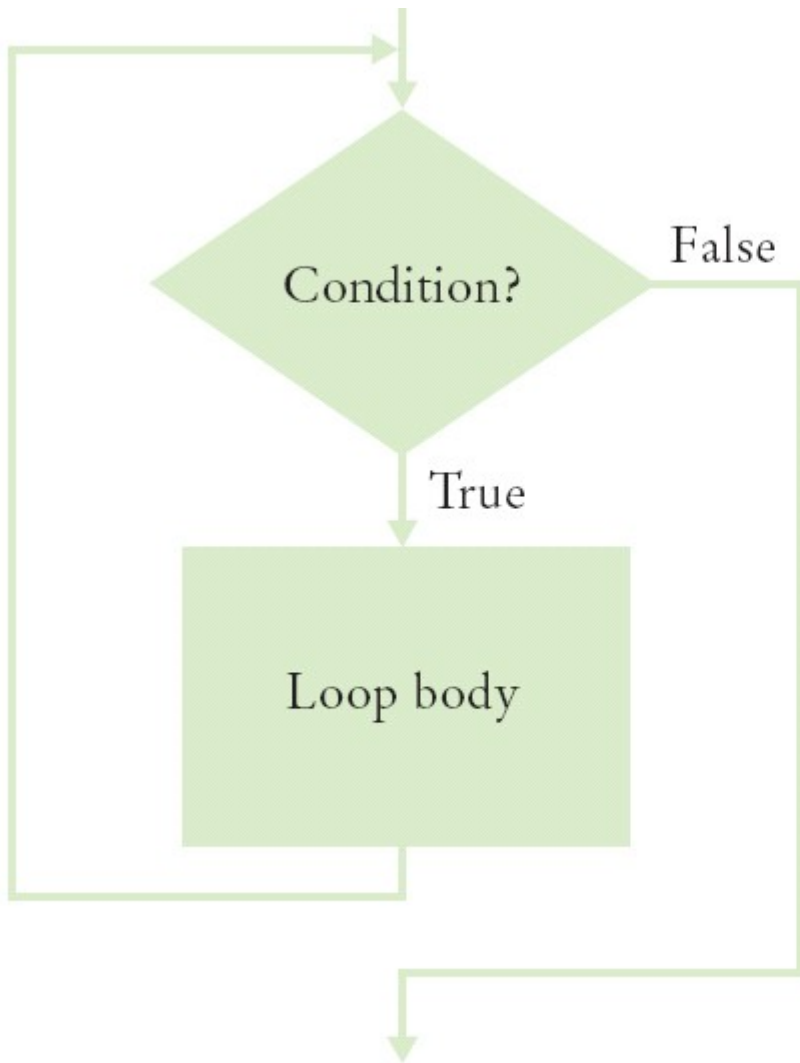
The do Loop

In order to have a different, “error” prompt that the user sees only on *invalid* input, the initial prompt and input would be before a **while** loop:

```
int value = 0;
printf("Enter a value < 100: ");
scanf("%d ", &value);
while (value >= 100) {
    printf("Sorry, that is larger than 100\n");
    printf("Try again: ");
    scanf("%d ", &value);
}
```

Notice what happens when the user gives valid input on the first attempt: nothing – good.

Flowcharts for the `while` Loop and the `do` Loop



Processing Input – When and/or How to Stop?

- We need to know, when getting input from a user, when they are done.
- One method is to use a sentinel
or more correctly choose a *value* whose meaning is STOP!
- As long as there is a known range of valid data points, we can use a value not in it.

Processing Input – When and/or How to Stop?

- We will write code to calculate the average of some salary values input by the user.

How many will there be?

That is the problem. We can't know.

But we can use a *sentinel value*, as long as we tell the user to use it, to tell us when they are done.

- Since salaries are never negative, we can safely choose -1 as our sentinel value.

Processing Input – When and/or How to Stop?

- In order to have a value to test, we will need to get the first input before the loop. The loop statements will process each non-sentinel value, and then get the next input.
- Suppose the user entered the sentinel value as the first input. Because averages involve division by the count of the inputs, we need to protect against dividing by zero. Using an **if-else** statement will do.

The Complete Salary Average Program

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double sum = 0.0;
    int count = 0;
    double salary = 0.0;
    // get all the inputs
    printf("Enter salaries, -1 to finish: ");
    while (salary != -1.0) {
        scanf("%lf", &salary);
        if (salary != -1.0) {
            sum = sum + salary;
            count++;
        }
    }
}
```


The Complete Salary Average Program

```
// process and display the average
if (count > 0) {
    double average = sum / count;
    printf("Average salary: %lf\n", average);
} else {
    printf("No data\n");
}

return EXIT_SUCCESS;
}
```

A program run:

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

The Loop and a Half Problem and the `break` Statement

Those same programmers who dislike loops that are controlled by a `bool` variable have another reason: the actual test for loop termination is in the *middle* of the loop. Again it is not really a top or bottom test.

This is called a loop-and-a-half.

The Loop and a Half Problem and the `break` Statement

If we test for a failed read, we can stop the loop *at that point*:

```
#include <stdio.h> //printf
#include <stdbool.h> //true

int main()
{
    int value = 0;
    while (true) {
        scanf("%d", &value);
        printf("%d--\n", value);
        if (value == -1) {
            break;
        }
    }
    return 0;
}
```

The **`break`** statement breaks out of the enclosing loop, independent of the loop condition.

Nested Loops

- Nested loops are used mostly for data in tables as rows and columns.
- The processing across the columns is a loop, as you have seen before, “nested” inside a loop for going down the rows.
- Each row is processed similarly so design begins at that level. After writing a loop to process a generalized row, that loop, called the “inner loop,” is placed inside an “outer loop.”

Nested Loops

Write a program to produce a table of powers.
The output should be something like this:

x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
...
10	100	1000	10000

Nested Loops

- The first step is to solve the “nested” loop.
- There are four columns and in each column we display the power. Using x to be the number of the row we are processing, we have (in pseudo-code):

```
for n from 1 to 4
{
    print  $x^n$ 
}
```

- You would test that this works in your code before continuing. If you can't correctly print one row, why try printing lots of them?

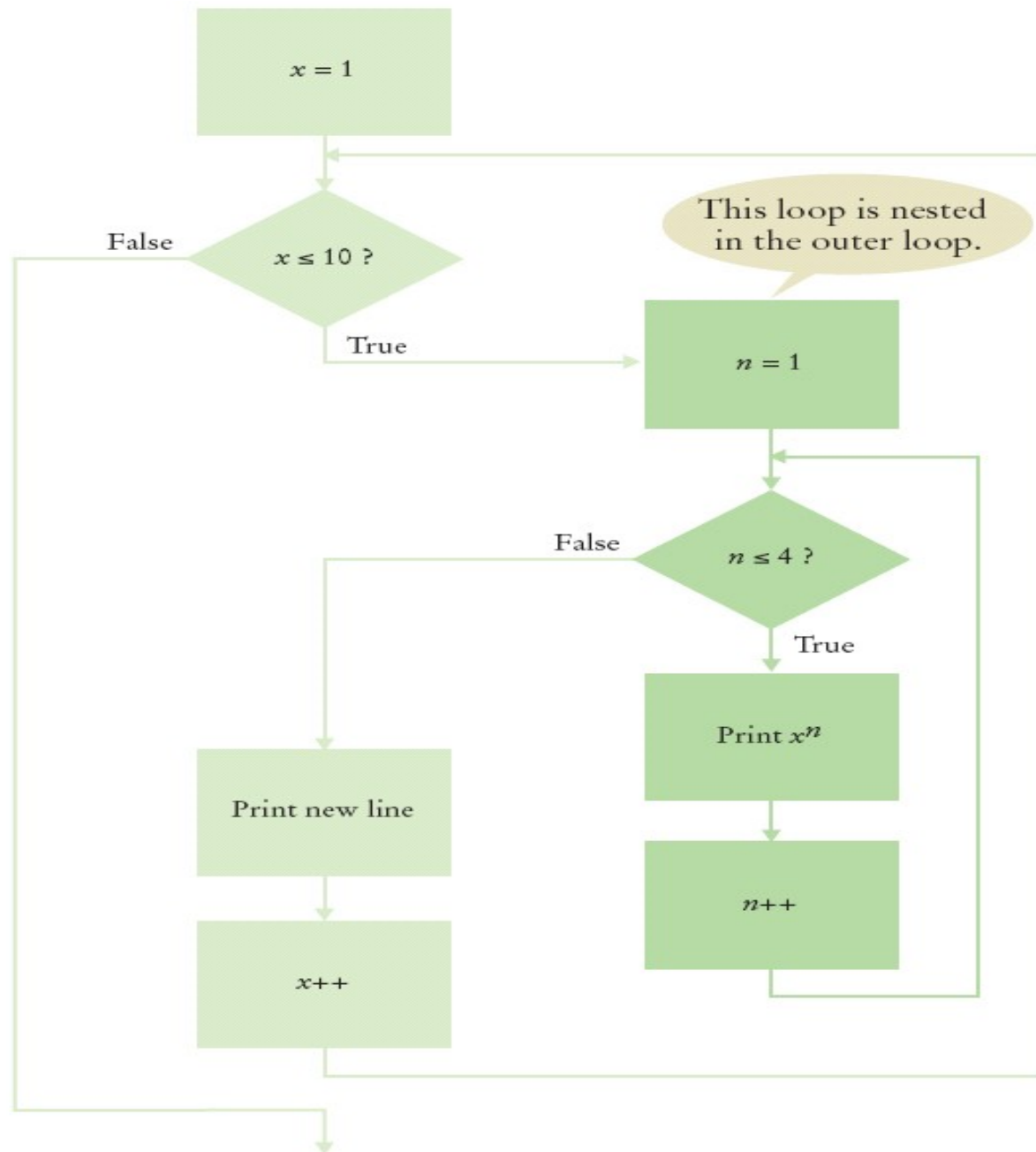
Nested Loops

Now, putting the inner loop into the whole process we have:

(don't forget to indent, nestedly)

```
print table header
for x from 1 to 10
{
    print table row
    print endl
}
```

Nested Loops



The Complete Program for Table of Powers

```
int main()
{
    const int NMAX = 4;
    const double XMAX = 10;

    // Print table header
    for (int n = 1; n <= NMAX; n++) {
        printf("%10d ", n);
    }
    printf("\n");
    for (int n = 1; n <= NMAX; n++) {
        printf("%9s ", "x");
    }
    printf("\n\n");
}
```

The Complete Program for Table of Powers

```
// Print table body
for (double x = 1; x <= XMAX; x++) {
    // Print table row
    for (int n = 1; n <= NMAX; n++) {
        printf("%10.01f ", pow(x, n));
    }
    printf("\n");
}

return EXIT_SUCCESS;
}
```

The program run would be:

1	2	3	4
x	x	x	x
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625

More Nested Loop Examples

The loop variables can have a value relationship. In this example the inner loop depends on the value of the outer loop.

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= i; j++) {  
        printf("*");  
    }  
    printf("\n");  
}
```

The output will be:

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

More Nested Loop Examples

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= i; j++)  
        printf("*");  
    printf("\n");  
}
```

j is *each* line's length,
which is different for each line, and
depends on the current line number, *i*

i represents the
row number or
the line number

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

More Nested Loop Examples

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= i; j++)  
        printf("*");  
    printf("\n");  
}
```

j is *each* line's length,
which is different for each line, and
depends on the current line number, *i*

j stops at: *i*
1

when *i* is: *i* 1

i represents the
row number or
the line number

More Nested Loop Examples

```
for (i = 1; i <= 4; i++)  
    for (j = 1; j <= i; j++)  
        cout << "*";  
cout << endl;
```

j is *each* line's length,
which is different for each line and
depends on the current line number, *i*

j stops at: *i*
1

when *i* is: *i* 1

*

i represents the
row number or
the line number

More Nested Loop Examples

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= i; j++)  
        printf("*");  
    printf("\n");  
}
```

j is *each* line's length,
which is different for each line. and
depends on the current line number, *i*

j stops at: *i*
1

when *i* is: *i* 1
 i 2

```
*  
* *
```

i represents the
row number or
the line number

More Nested Loop Examples

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= i; j++)  
        printf("*");  
    printf("\n");  
}
```

j is *each* line's length,
which is different for each line, and
depends on the current line number, *i*

j stops at: *i i i*
 1 2 3

when *i* is: *i 1*
 i 2
 i 3

i represents the
row number or
the line number

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


More Nested Loop Examples

```
for (int i = 1; i <= 4; i++) {  
    for (int j = 1; j <= i; j++)  
        printf("*");  
    printf("\n");  
}
```

j is *each* line's length,
which is different for each line, and
depends on the current line number, *i*

j stops at: *i* *i* *i* *i*
 1 2 3 4

when *i* is: *i* 1
 i 2
 i 3
 i 4

i represents the
row number or
the line number

*			
*	*		
*	*	*	
*	*	*	*

More Nested Loop Examples

In this example, the loop variables are still related, but the processing is a bit more complicated.

```
for (int i = 1; i <= 3; i++) {  
    for (int j = 1; j <= 5; j++) {  
        if ((i + j) % 2 == 0) {  
            printf("*");  
        } else {  
            Printf(" ");  
        }  
        printf("\n");  
    }  
}
```

The output will be:

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

Simulations

A simulation program uses the computer to simulate an activity in the real world (or in an imaginary one).

Simulations

- Simulations are commonly used for
 - Predicting climate change
 - Analyzing traffic
 - Picking stocks
 - Many other applications in science and business

Randomness for Reality (Simulating)

- Programmers must model the “real world” at times.
- Consider the problem of modeling customers arriving at a store.

Do we know the rate?

Does anyone?

Randomness for Reality (Simulating)

To accurately model customer traffic, you want to take that random fluctuation into account.

How?

The `rand` Function

The C library has a random number generator:

`rand()`

The `rand` Function

`rand` is defined in the `stdlib` header

Calling `rand` yields a random integer
between 0 and `RAND_MAX`

(The value of `RAND_MAX` is implementation dependent)

The `rand` Function

Calling `rand` again yields a different random integer

Very, very, very rarely it might be the same random integer again.

(That's OK. In the real world this happens.)

The `rand` Function

`rand` picks from a very long sequence of numbers that don't repeat for a long time.

But they do eventually repeat.

These sorts of “random” numbers are often called *pseudorandom numbers*.

The `rand` Function

`rand` uses only one pseudorandom number sequence and it always starts from the same place.

The `rand` Function

When you run your program again on another day, the call to `rand` will start with:

*the **same** random number!*

Is it very “real world” to use the same sequence over and over?

No, but it’s really nice for testing purposes.

but...

Seeding the `rand` Function

You can “seed” the random generator to indicate where it should start in the pseudorandom sequence

Calling `srand` sets where `rand` starts

`srand` is defined in the `stdlib.h` header

Seeding the rand Function

But what value would be different every *time* you run your program?

(hint)



How about the time?

Seeding the `rand` Function

You can obtain the system time.

Calling `time(NULL)` gets the current time

Note the `NULL`. It is required.

`time` is defined in the `time` header

Seeding the rand Function

Calling `srand` sets where `rand` starts.

Calling `time(NULL)` gets the current time.

So, to set up for “really, really random”
random numbers on each program run:

```
srand(time(NULL)) ; // seed rand()
```

(Well, as “really random” as we can hope for.)

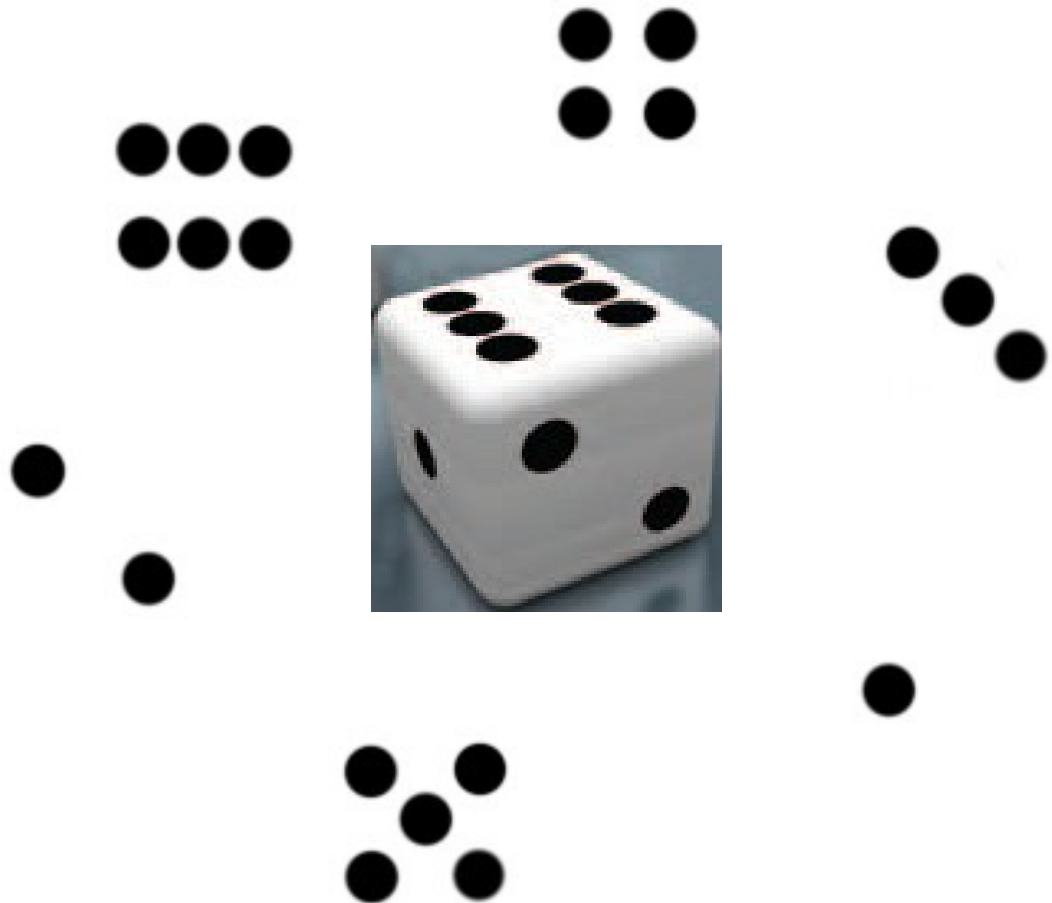
Modeling Using the `rand` Function

Let's model a pair of dice,



Modeling Using the rand Function

What are the numbers on one die?



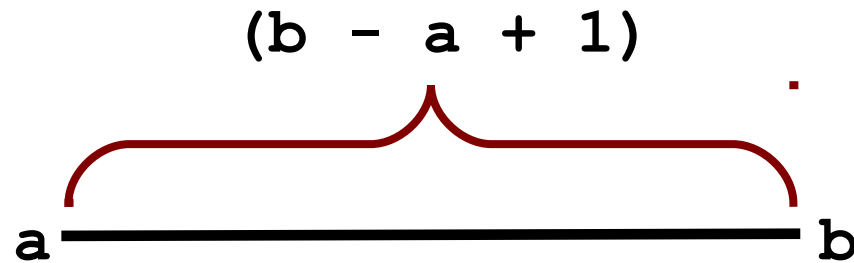
Modeling Using the `rand` Function

What are the bounds of the range of numbers on one die?
1 and 6 (inclusive)



We want a value randomly between those endpoints
(inclusively)

Modeling Using the rand Function

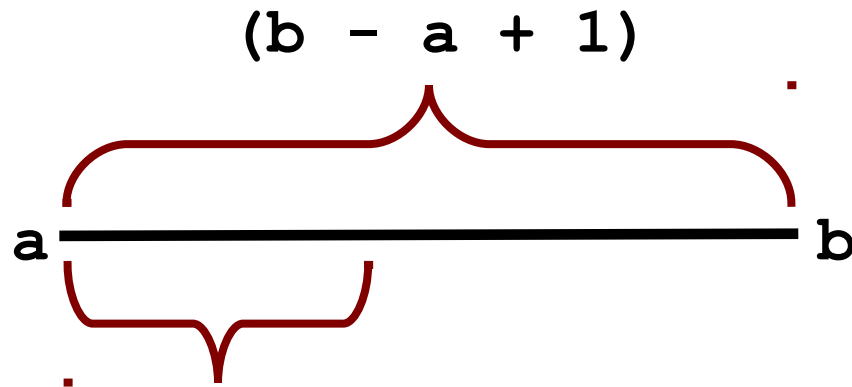


Given two endpoints,
a and **b**, recall there are

$$(b - a + 1)$$

values between **a** and **b**,
(including the bounds themselves).

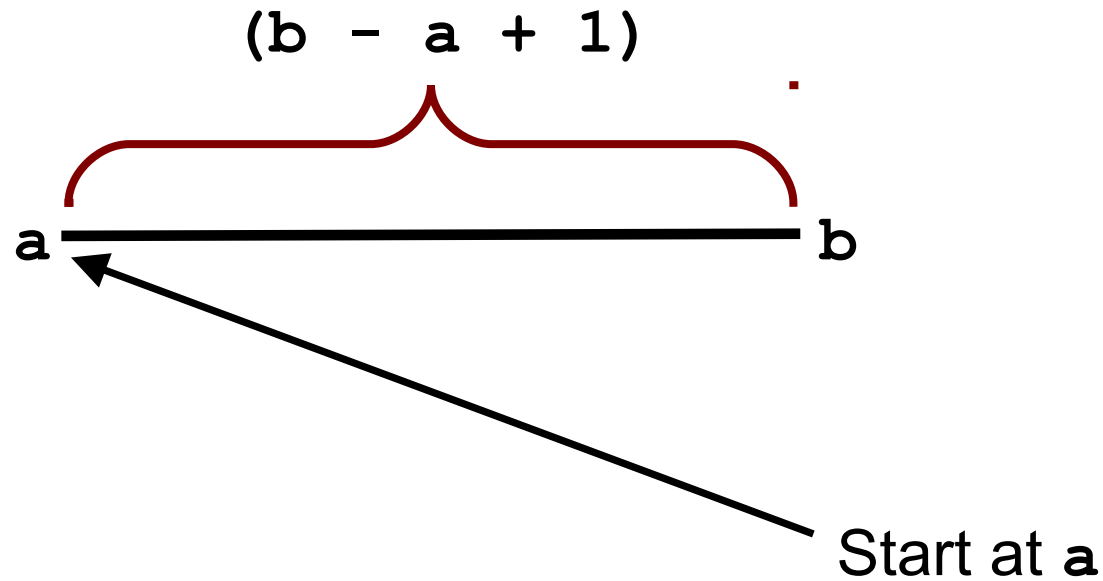
Modeling Using the `rand` Function



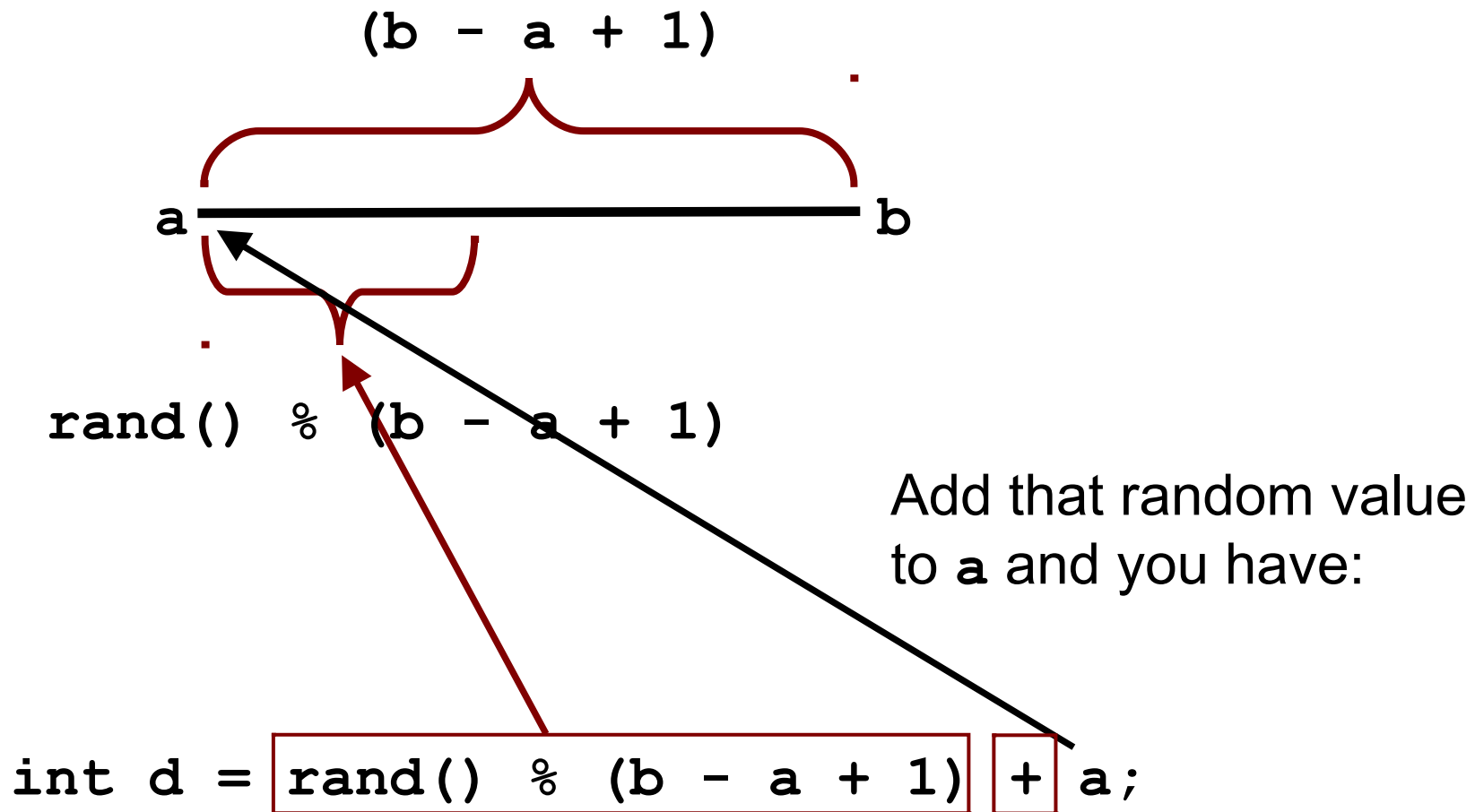
`rand() % (b - a + 1)`

Obtain a random value
between 0 and $b - a$
by using the `rand()` function

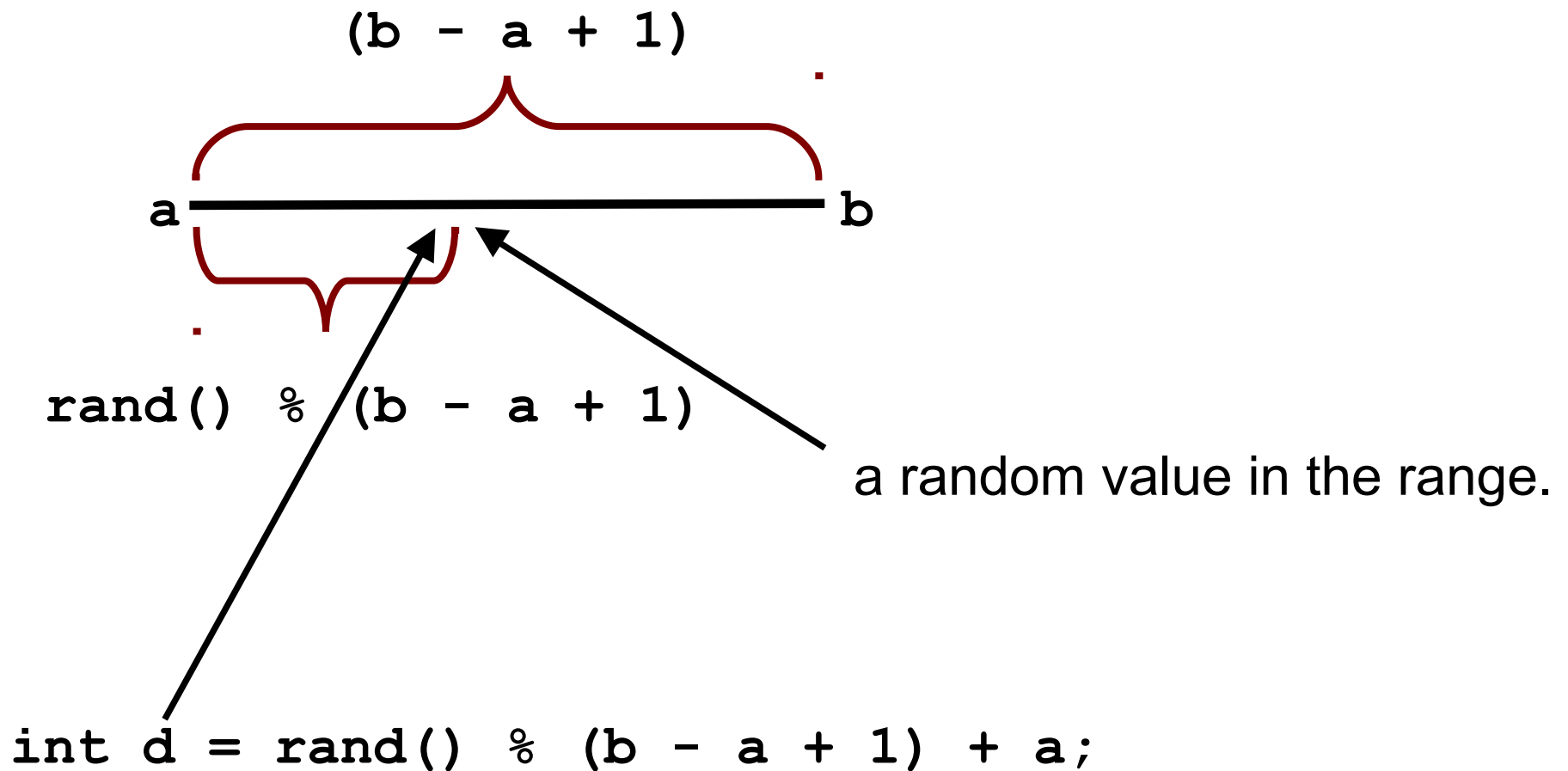
Modeling Using the rand Function



Modeling Using the rand Function



Modeling Using the rand Function



Modeling Using the `rand` Function



Using 1 and 6 as the bounds
and
modeling for two dice,
running for 10 tries,

we have:

Modeling Using the rand Function

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand(time(NULL));

    for (int i = 1; i <= 10; i++) {
        int d1 = rand() % 6 + 1;
        int d2 = rand() % 6 + 1;
        printf("%d %d\n", d1, d2);
    }
    return EXIT_SUCCESS;
}
```

One of many different
program runs:

5	1
2	1
1	2
5	1
1	2
6	4
4	4
6	1
6	3
5	2



End Chapter Four