Computer Programming

Basic Control Flow - Loops

Adapted from C++ for Everyone and Big C++ by Cay Horstmann, John Wiley & Sons

Objectives

- To learn about the three types of loops:
  - **`while`**
  - **`for`**
  - **`do`**
- To avoid infinite loops and off-by-one errors
- To understand nested loops and sentinel.

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
```

The `while` Loop

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

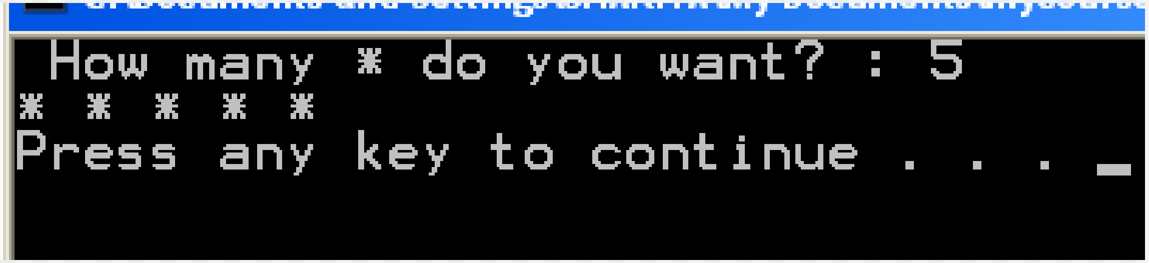
The *condition* is some kind of test
(the same as it was in the **`if`** statement in Chap. 3)

The `while` Loop

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

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

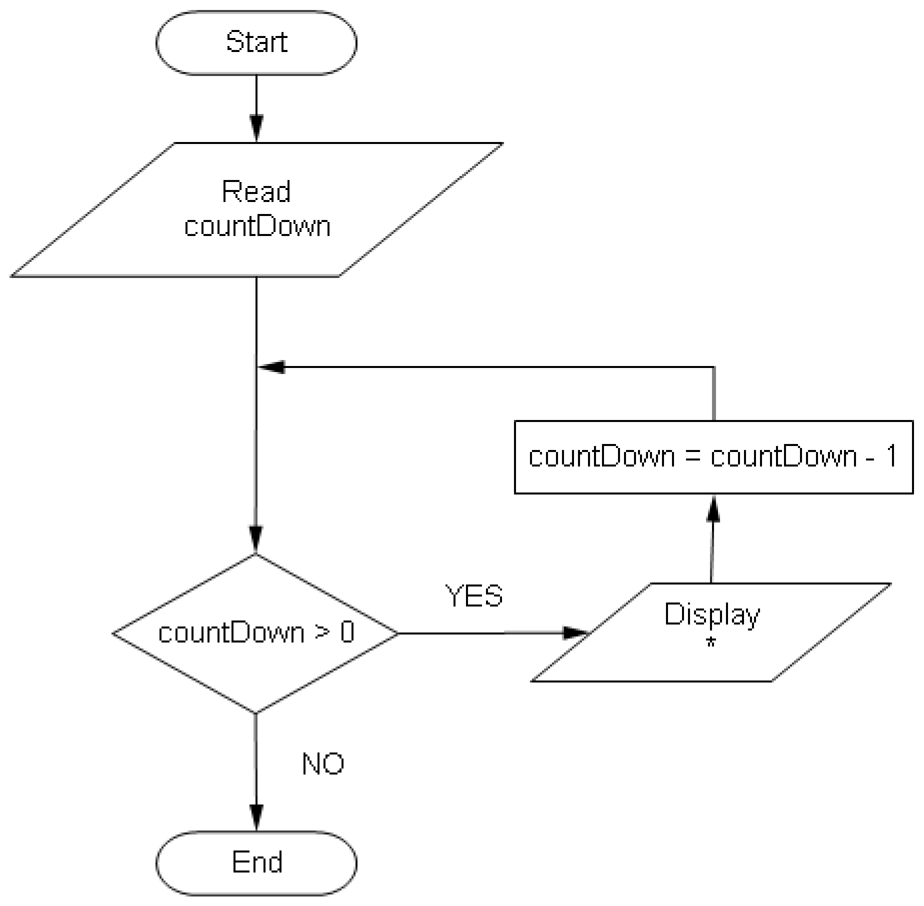
The `while` Loop



```
 How many * do you want? : 5
* * * * *
Press any key to continue . . . _
```

```
1.0 START
2.0 Read input from user, countDown.
3.0 IF the countDown is more than 0 THEN
        3.1 Display *
        3.2 Decrement countDown by 1
        3.3 Repeat step 3.0.
        ELSE
        3.4 Go to Step 4.0.
4.0 END.
```

The `while` Loop

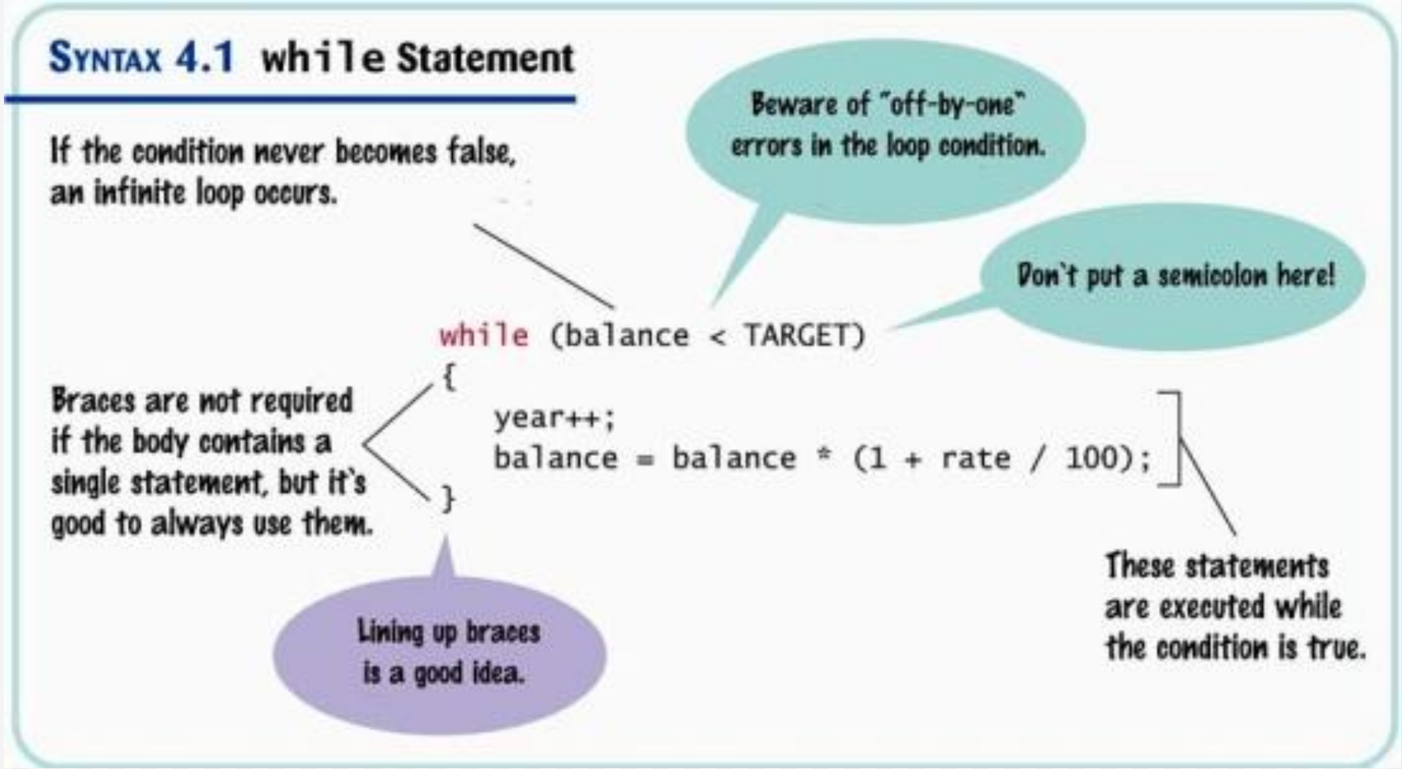


```
cin >> countDown;

     while( countDown > 0)
     {
              cout << "* " ;

              countDown--;
     }
```

The `while` Loop

- When doing something repetitive, we want to know when we are done.
- Example:
  - I want to get at least $20,000
  - So we set → **`balance >= TARGET`**
- But the while loop thinks the opposite:
  - How long I am allowed to keep going?
  - So the correct condition :
    - **`while (balance < TARGET)`**

The `while` Loop



Syntax 4.1 while Statement

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

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

Don't put a semicolon here!

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

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

Lining up braces is a good idea.

These statements are executed while the condition is true.

Using a Loop to Solve the Investment Problem.

The algorithm for an investment problem:

1. Start with a year value of 0 and a balance of $10,000.
2. <u>**Repeat**</u> the following steps <u>**while the balance is less than $20,000**:</u>
   - Add 1 to the year value.
   - Multiply the balance value by 1.05 (a 5 percent increase).
3. Report the final year value as the answer.

"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
Updating the **`balance`** variable using a **`const`** for the **`RATE`**

```
year++;
balance = balance * (1 + RATE / 100);
```

Using a Loop to Solve the Investment Problem.

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

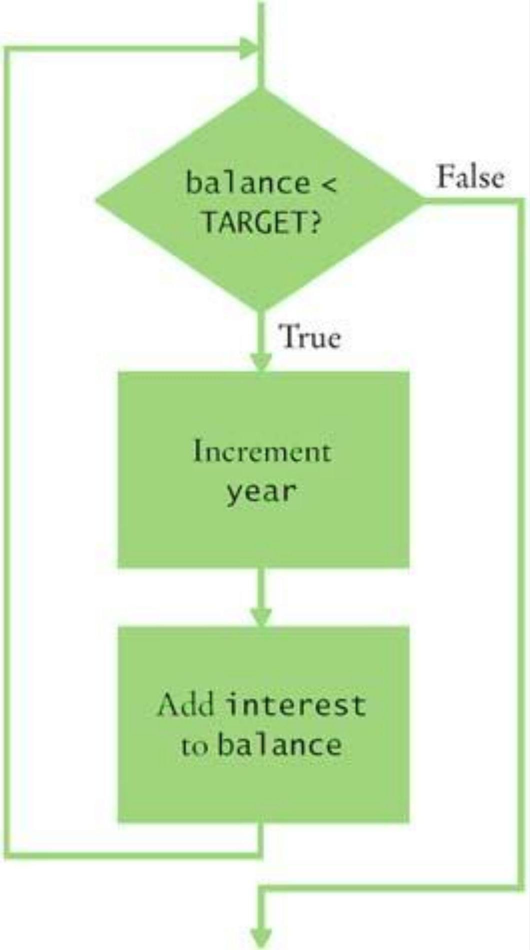
```
(balance < TARGET)
```

Using a Loop to Solve the Investment Problem.

Here is the complete **while** statement:

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

Flowchart of the Investment Calculation's `while` Loop



The Complete Investment Program

```
#include <iostream>
using namespace std;

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++;
      balance = balance * (1 + RATE / 100);
   }

   cout << "The investment doubled after "
        << year << " years." << endl;

   return 0;
}
```

Program Run

1. Check the loop condition

```
balance = 10000
year    = 0
```

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

Program Run

1. Check the loop condition

```
balance = 10000
year    = 0
```

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

The condition is true

Program Run

1. Check the loop condition

```
balance = 10000
year    = 0
```

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

The condition is true

2. Execute the statements in the loop

```
balance = 10500
year    = 1
```

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

Program Run

```
3. Check the loop
   condition again

   balance = 10500
   year    = 1
```

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

The condition is still true

Program Run

```
3. Check the loop
   condition again

   balance = 10500
   year    = 1
```

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

The condition is still true

```
4. Execute the
   statements in
   the loop

   balance = 11000
   year    = 2
```

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

Program Run

```
3. Check the loop
   condition again

   balance = 11000
   year    = 2
```

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

The condition is still true

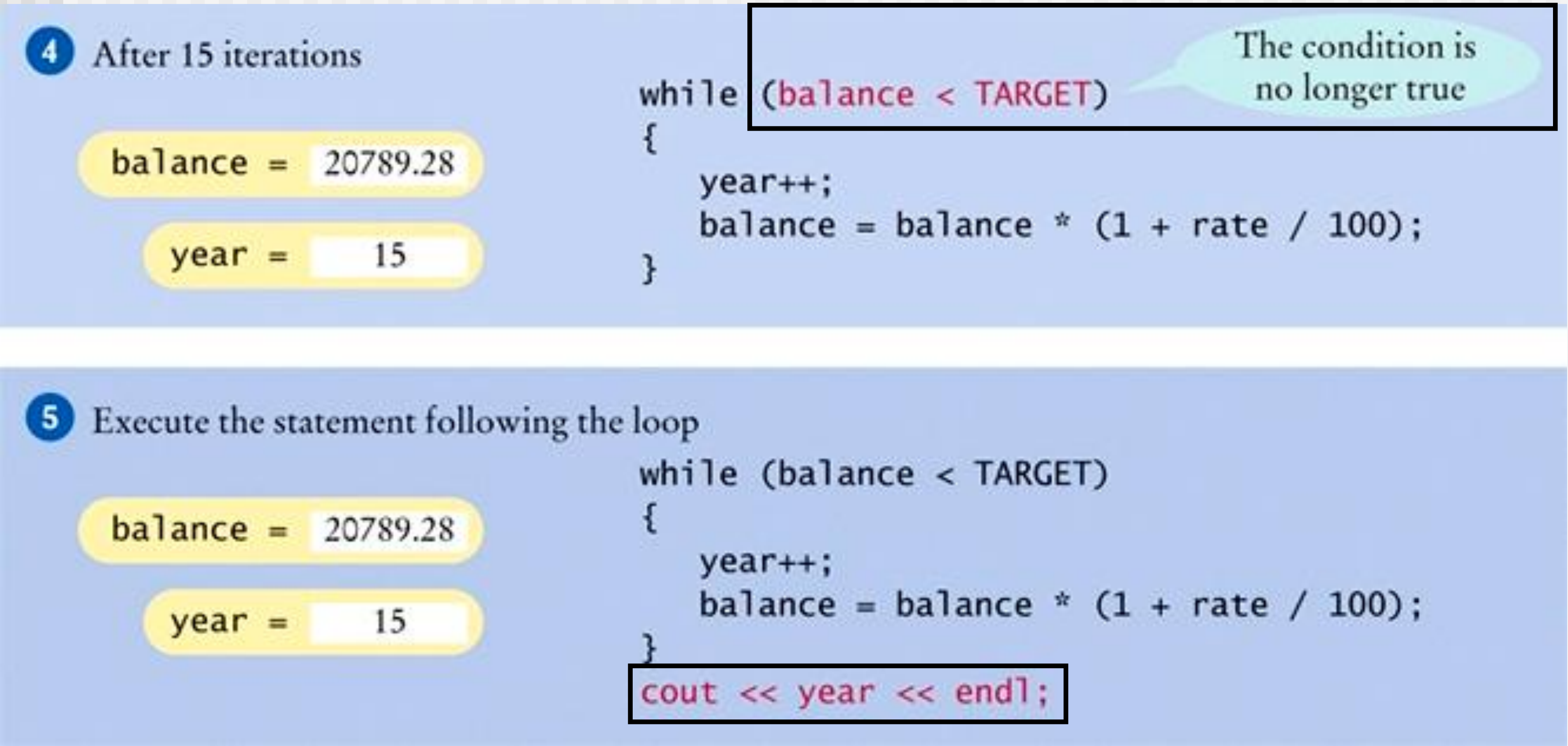
```
4. Execute the
   statements in
   the loop

   balance = 11500
   year    = 3
```

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

Program Run

…This process continues
for 15 iterations…

Program Run



4 After 15 iterations

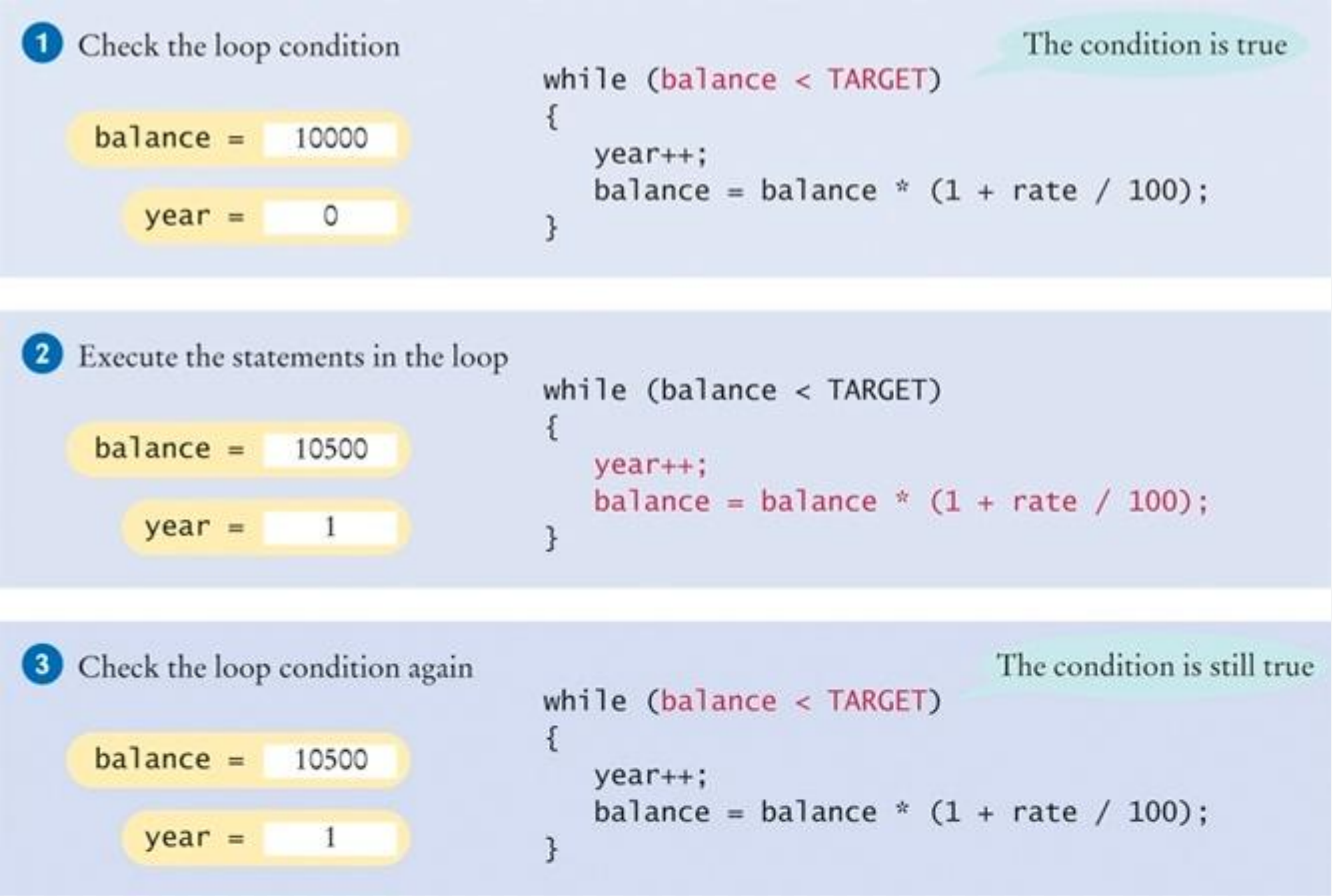
```
while (balance < TARGET)
{
   year++;
   balance = balance * (1 + rate / 100);
}
```

5 Execute the statement following the loop

```
while (balance < TARGET)
{
   year++;
   balance = balance * (1 + rate / 100);
}
cout << year << endl;
```

The final output indicates that the investment doubled in 15 years.

Program Run



1 Check the loop condition

The condition is true

balance = 10000

year = 0

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

2 Execute the statements in the loop

balance = 10500

year = 1

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

3 Check the loop condition again

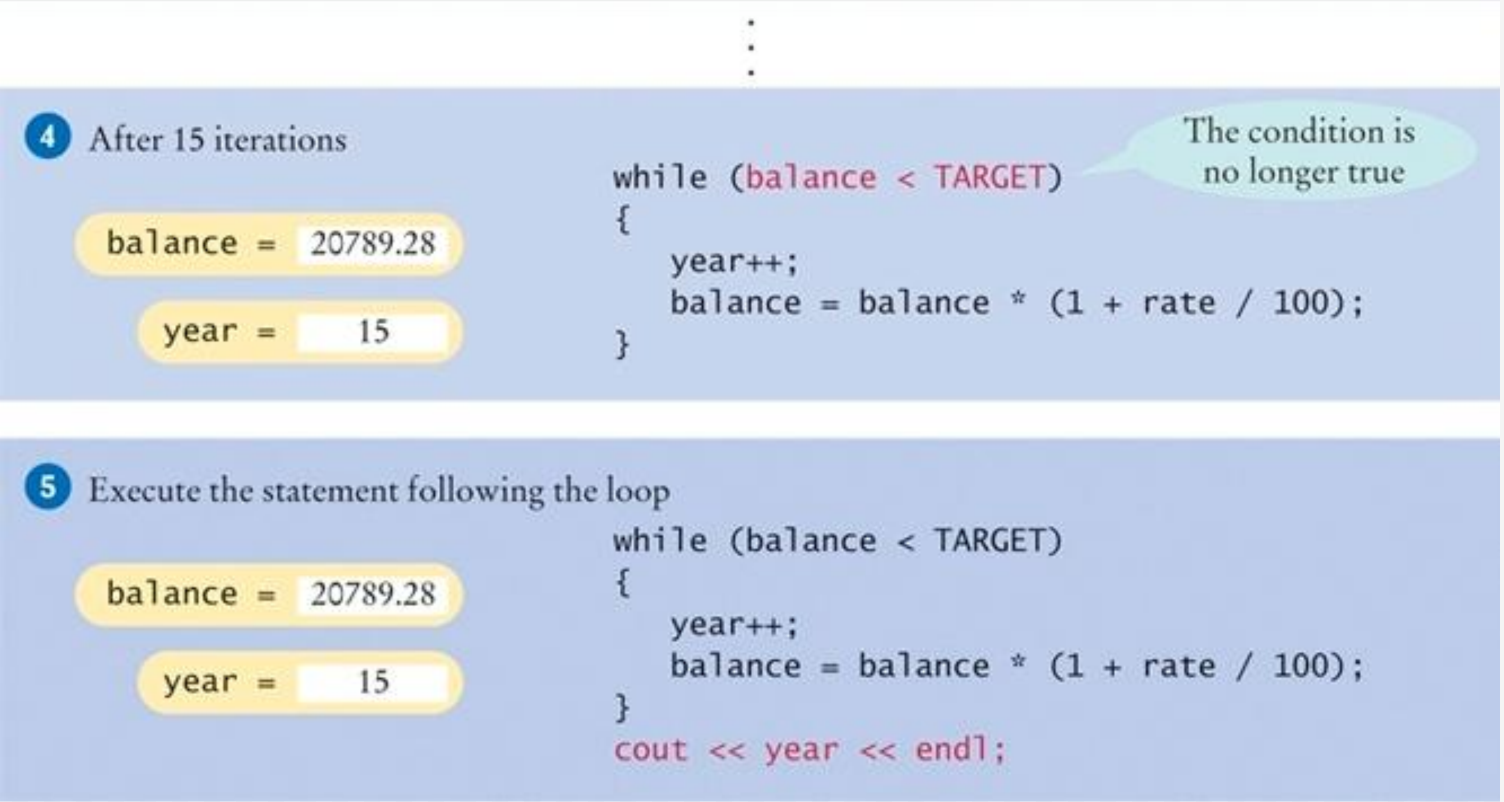
The condition is still true

balance = 10500

year = 1

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

Program Run



4 After 15 iterations

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

5 Execute the statement following the loop

```
while (balance < TARGET)
{
   year++;
   balance = balance * (1 + rate / 100);
}
cout << year << endl;
```

More `while` Examples

Skip the examples?

NO YES

More `while` Examples

For each of the following, do a hand-trace

Example of Normal Execution

`while` loop to hand-trace

```
i = 5;
while (i > 0)
{
   cout << 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)
{
    cout << i << " ";
    i--;
}
```

The output

```
1 2 3 4 5
correct?
OR

5 4 3 2 1
```

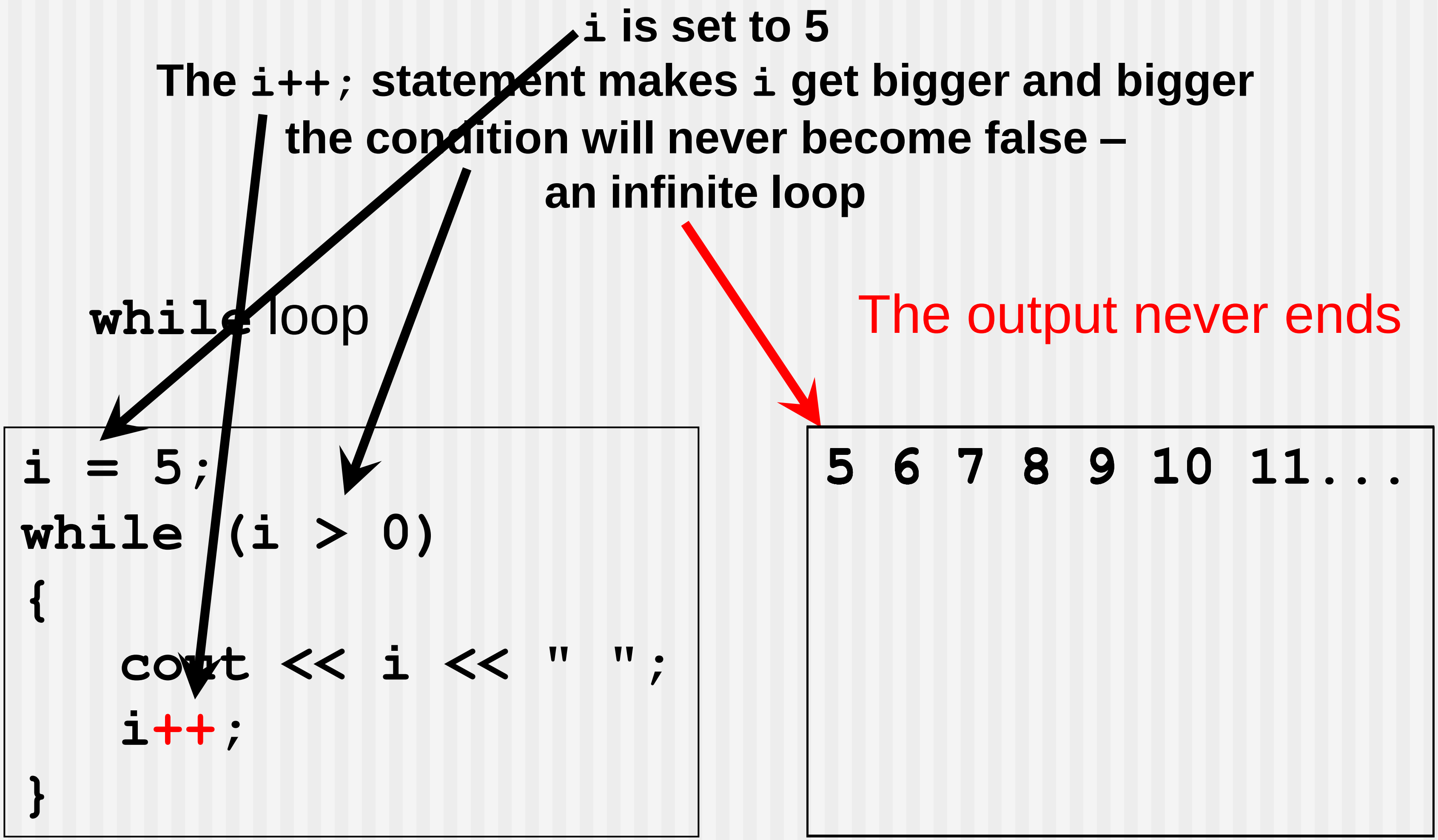
Example of a Problem – An Infinite Loop

`while` loop to hand-trace

What is the output?

```
i = 5;
while (i > 0)
{
   cout << i << " ";
   i++;
}
```

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)
{
   cout << 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)
{
   cout << i << " ";
   i--;
}
```

What is the output?

Another Normal Execution – No Errors

`while` loop

```
i = 5;
while (i > 5)
{
   cout << 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)
{
   cout << 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

What is the output?

```
i = 5;
while (i < 0)
{
    cout << i << " ";
    i--;
}
```

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)
{
    cout << 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);
{
    cout << i << " ";
    i--;
}
```

What is the output?

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);
{
    cout << 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 – Infinite Loops

Another way to cause an infinite loop:
Typing on “autopilot”

Typing **`++`** when you meant to type **`--`**
is a real problem, especially when it’s 3:30 am!

```
year = 20;
while (year > 0)
{
   balance = balance * (1 + RATE / 100);
   year++;
}
```

A Not Really Infinite Infinite Loop

- Due to what is called “wrap around”, the previous loop ***will*** end.
- At some point the value stored in the **`int`** variable gets to the largest representable positive integer. When it is incremented, the value stored “wraps around” to be a negative number.

That definitely stops the loop!

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 – 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)

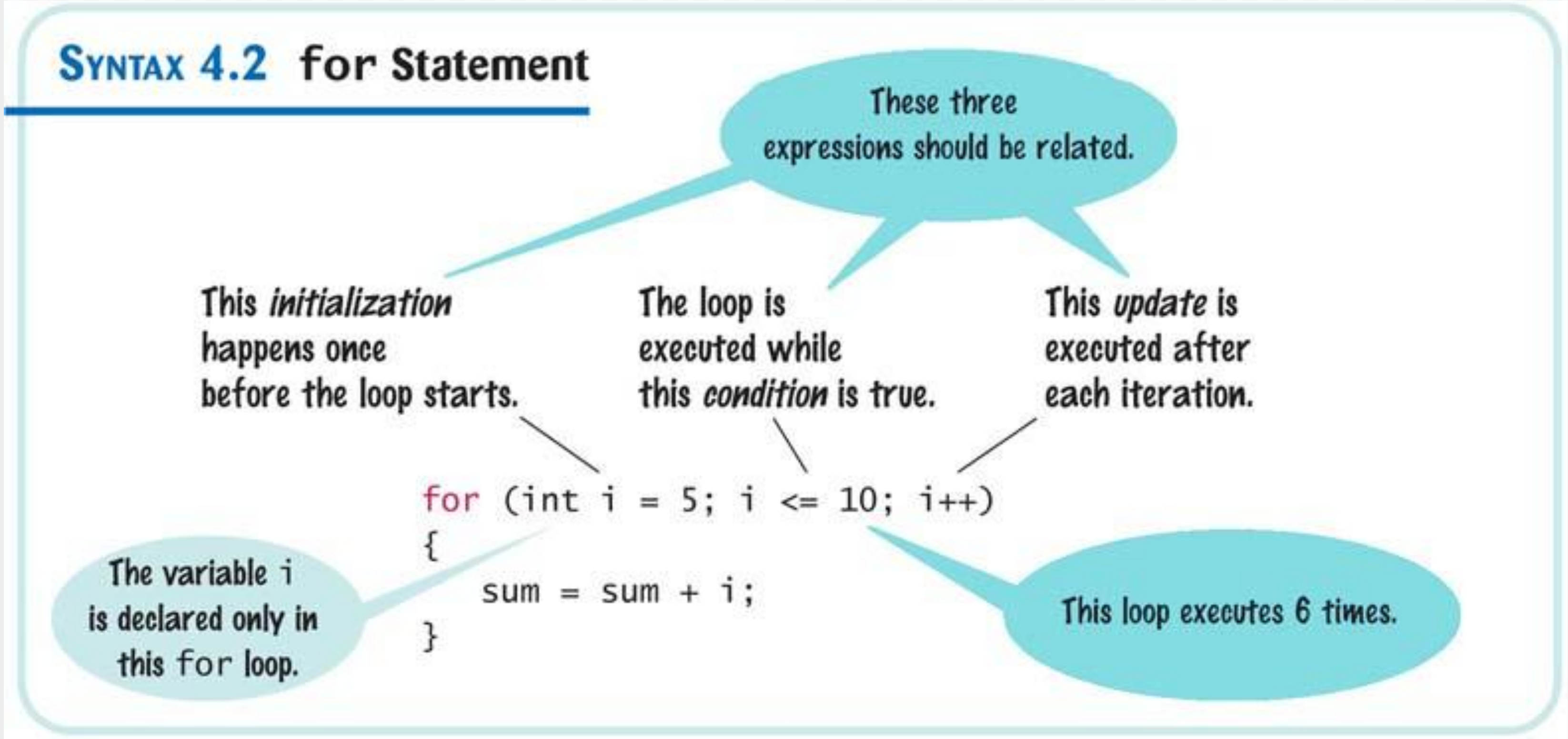
- But consider, did you really mean:

  “…to have *at least* doubled…”

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

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

The `for` Loop



SYNTAX 4.2 for Statement

These three expressions should be related.

This *initialization* happens once before the loop starts.

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

This *update* is executed after each iteration.

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

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

This loop executes 6 times.

The `for` Loop

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

The *check* is some kind of test (the same as the condition in the **`while`** loop)

It is usually used to cause the *statements* to happen a certain number of times.

The `for` Loop

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

The *statements* are repeatedly executed until the check is false.

The `for` Loop

```
for (initialization; check; 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 `for` Loop

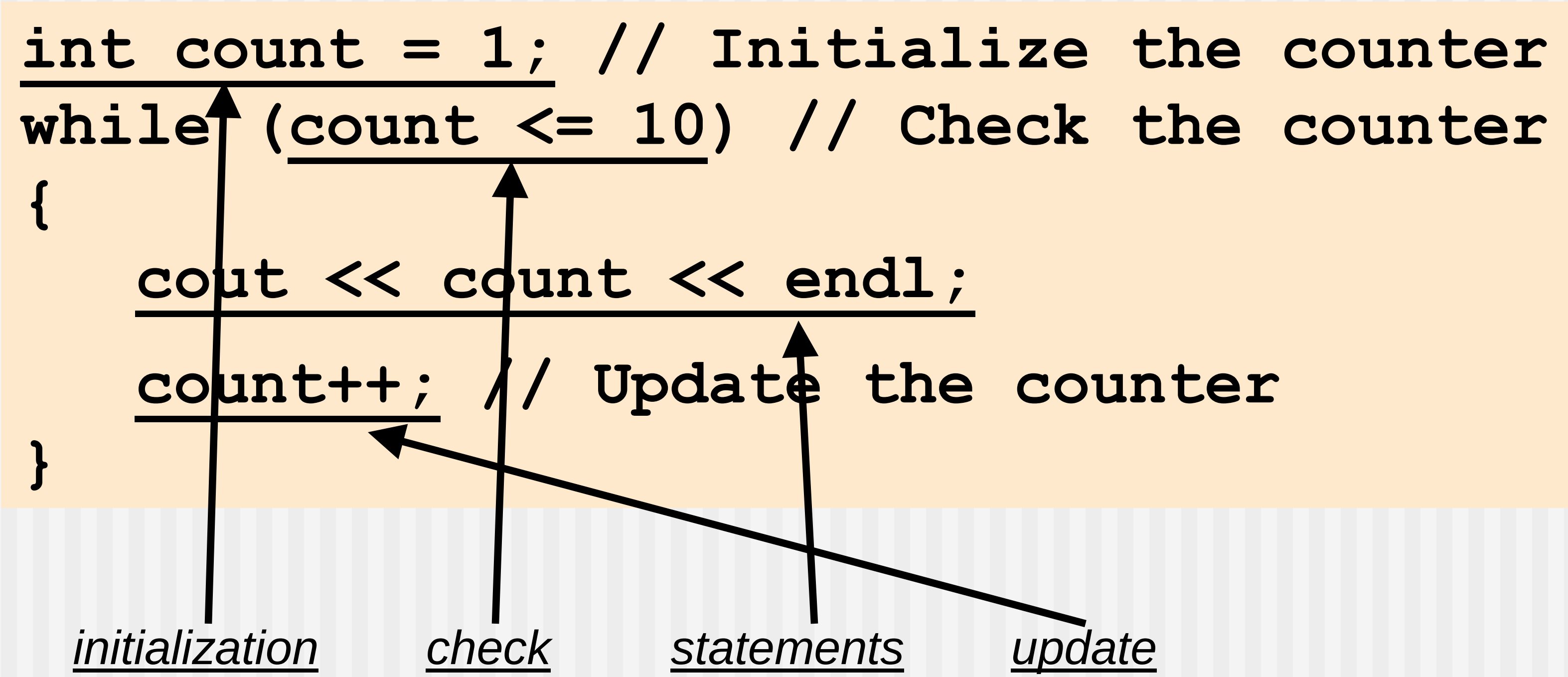
```
for (initialization; check; update)
{
    statements
}
```

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

Usually it’s incrementing or decrementing the loop variable.

The `for` Loop Is Better than `while` for Doing Certain Things

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

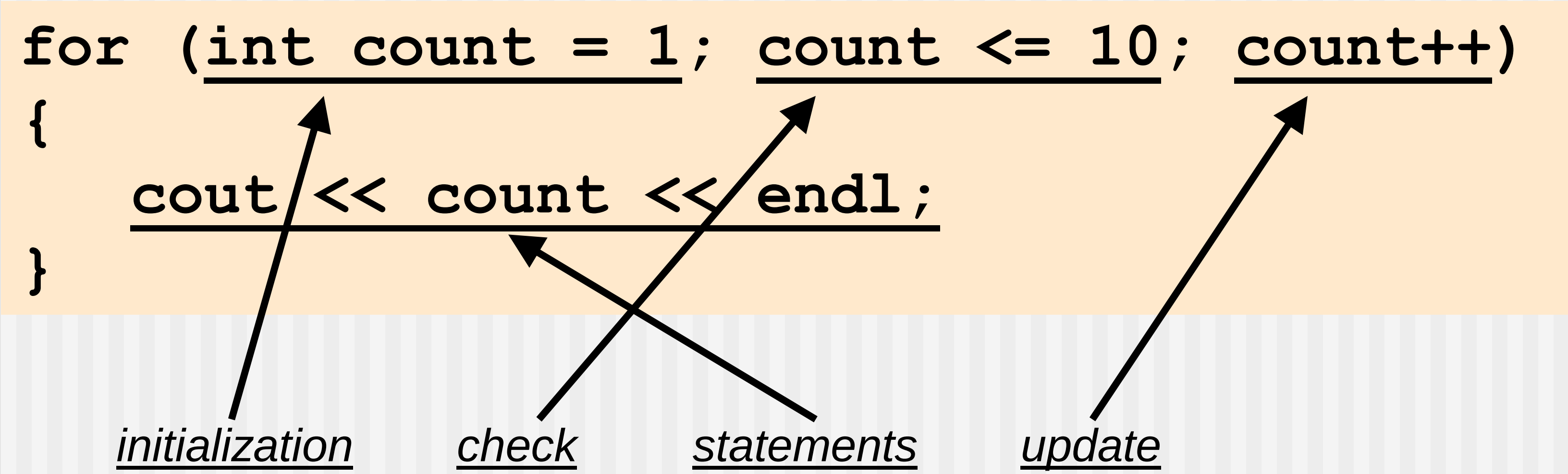


```
int count = 1; // Initialize the counter
while (count <= 10) // Check the counter
{
    cout << count << endl;
    count++; // Update the counter
}
```

initialization *check* *statements* *update*

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:



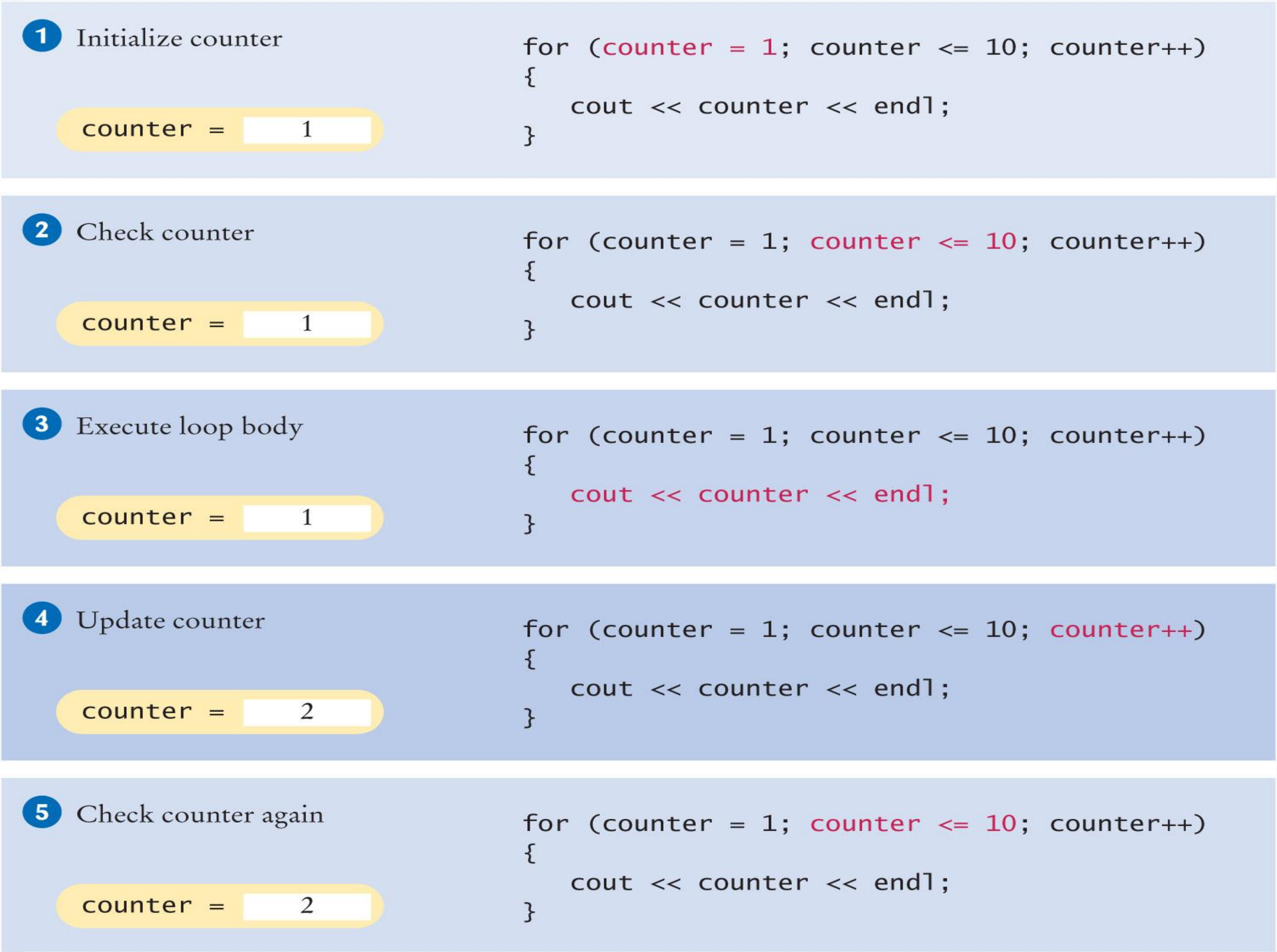
```
for (int count = 1; count <= 10; count++)
{
    cout << count << endl;
}
```

initialization *check* *statements* *update*

Execution of a `for` Statement

Consider this **for** statement:

```
int count;

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



1 Initialize counter

counter = 1

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

2 Check counter

counter = 1

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

3 Execute loop body

counter = 1

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

4 Update counter

counter = 2

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

5 Check counter again

counter = 2

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

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.

Solving a Problem with a `for` Statement

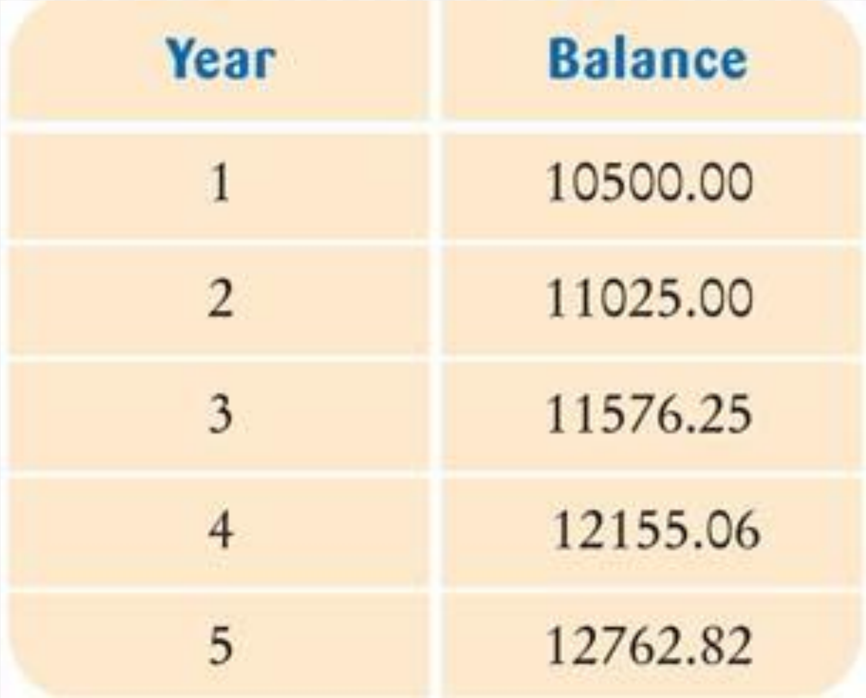
Earlier we determined the number of years it would take to (at least) double our balance.

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:

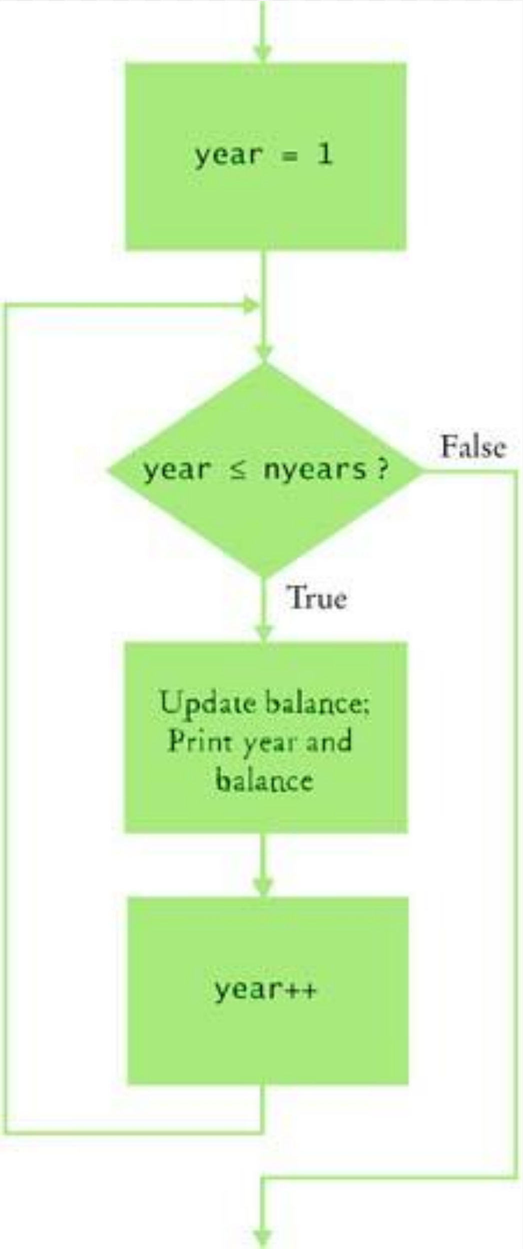


Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The `for` Loop

Flowchart of the investment calculation's `while` loop

Easily written 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 <iostream>
#include <iomanip>
using namespace std;

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

    cout << "Enter number of years: ";
    cin >> nyears;

    cout << fixed << setprecision(2);
    for (int year = 1; year <= nyears; year++)
    {
        balance = balance * (1 + RATE / 100);
        cout << setw(4) << year << setw(10) << balance << endl;
    }

    return 0;
}
```

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
```

More `for` Examples

For each of the following, do a hand-trace.

Example of Normal Execution

for loop to hand-trace

```
for (int i = 0;i <= 5;i++)
   cout << i << " ";
```

What is the output?

Example of Normal Execution

`for` loop

```
for (int i = 0;i <= 5;i++)
     cout << 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--)
   cout << 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--)
    cout << 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)
   cout << i << " ";
```

What is the output?

```
0 2 4 6 8
```

Example of Normal Execution – Taking Bigger Steps

for loop

```
for (int i = 0;
     i < 9;
     i += 2)
   cout << 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)
   cout << 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)
   cout << 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)
   cout << 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)
   cout << i << " ";
```

The output

```
1 2 4 8 16
```

The “step” can be multiplicative or any valid expression

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.

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.

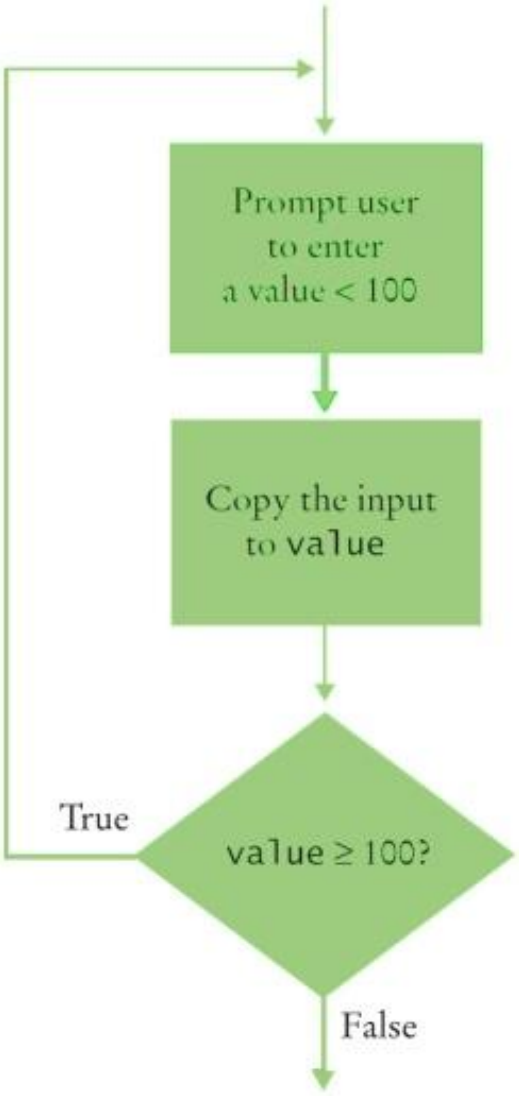
This also means that the **`do`** loop is the least used loop.

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;
do
{
   cout << "Enter a value < 100";
   cin >> value;
}
while (value >= 100);
```

In this form, the user sees the same prompt each time until the 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;
cout << "Enter a value < 100";
cin >> value;

while (value >= 100);
{
   cout << "Sorry, that is larger than 100\n"
        << "Try again: ";
   cin >> value;
}
```

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

Nested Loops

For each hour, 60 minutes are processed – a nested loop.

Nested Loops

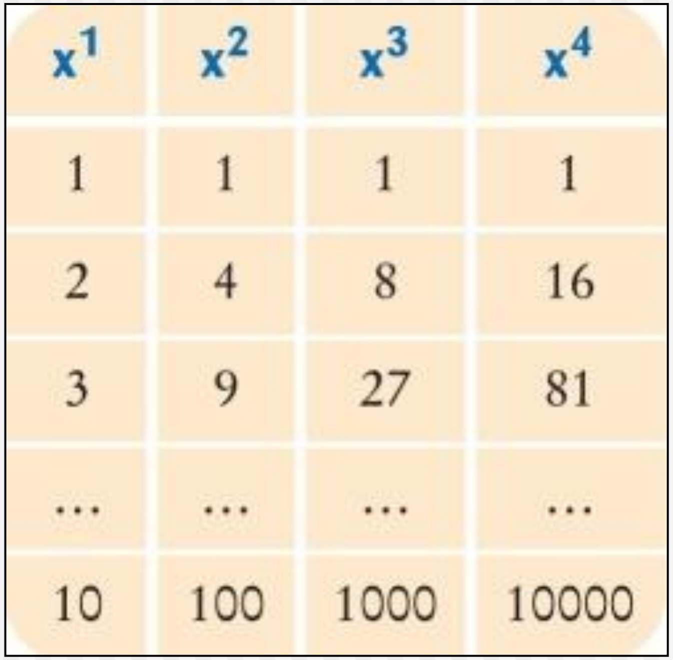
Nested loops are used mostly for data in tables as rows and columns.

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

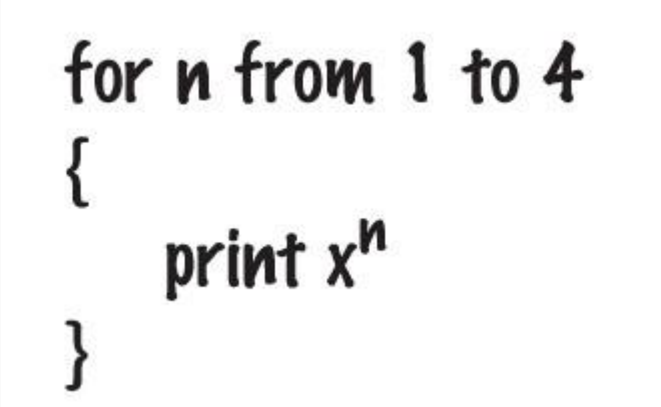
Write a program to produce a table of powers. The output should be something like this:



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):



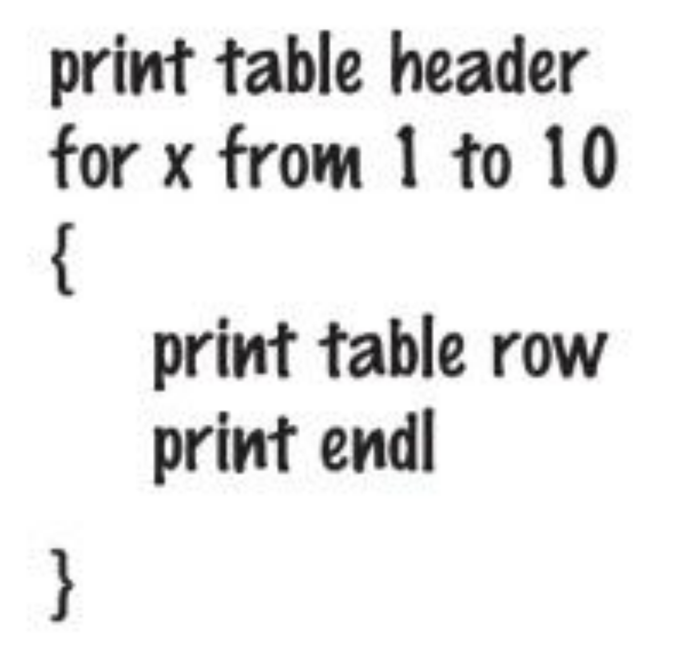
```
for n from 1 to 4
{
    print $x^n$
}
```

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

}
```

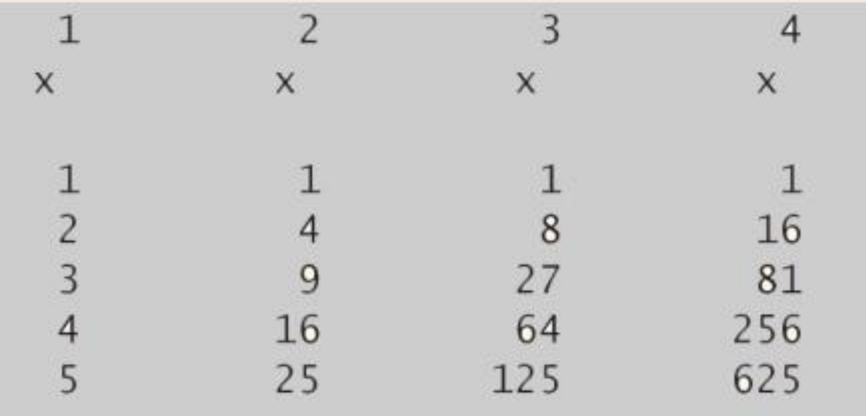
The Complete Program for Table of Powers

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

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

    // Print table header
    for (int n = 1; n <= NMAX; n++)
    {
        cout << setw(10) << n;
    }
    cout << endl;
    for (int n = 1; n <= NMAX; n++)
    {
        cout << setw(10) << "x "; // print x
    }
    cout << endl << endl;
```

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++)
        {
            cout << setw(10) << pow(x, n);
        }
        cout << endl;
    }

    return 0;
}
```

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

The output will be:

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

More Nested Loop Examples

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

`j` is the number of “columns” (or the line’s length), which depends on the line number, `i`

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

      line num. 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 (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 5; j++)
   {
      if (i + j % 2 == 0)
      { cout << "*"; }
      else { cout << " "; }
   }
   cout << endl;
}
```

The output will be:

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

Processing Input – When and/or How to Stop?

or be stopped!

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 hire a sentinel (as shown)

  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 from Lecture 3 will do.

The Complete Salary Average Program

```
#include <iostream>
using namespace std;

int main()
{
   double sum = 0;
   int count = 0;
   double salary = 0;

   // get all the inputs
   cout << "Enter salaries, -1 to finish: ";
   cin >> salary;
   while (salary != -1)
   {
      // process input
      sum = sum + salary;
      count++;

      // get next input
      cin >> salary;
   }
```

The Complete Salary Average Program

```
   // process and display the average
   if (count > 0)
   {
      double average = sum / count;
      cout << "Average salary: " << average << endl;
   }
   else
   {
      cout << "No data" << endl;
   }

   return 0;

}
```

A program run:

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

Using Failed Input for Processing

- Sometimes is it easier and a bit more intuitive to ask the user to “Hit Q to Quit” instead or requiring the input of a sentinel value.

- Sometimes picking a sentinel value is simply impossible – if any valid number is allowed, which number could be chosen?

Using Failed Input for Processing

- In the previous chapter we used **`cin.fail()`** to test if the most recent input failed.

- Note that if you intend to take more input from the keyboard after using failed input to end a loop, you must reset the keyboard with **`cin.clear()`**.

Using Failed Input for Processing

If we introduce a bool variable to be used to test for a failed input, we can use **`cin.fail()`** to test for the input of a ‘Q’ when we were expecting a number:

Using Failed Input for Processing

```
cout << "Enter values, Q to quit: ";
bool more = true;
while (more)
{
   cin >> value;
   if (cin.fail())
   {
      more = false;
   }
   else
   {
      // process value here
   }
}
cin.clear() // reset if more input is to be taken
```

Using Failed Input for Processing

Using a **`bool`** variable in this way is disliked by many programmers.

Why?

`cin.fail` is set when **>>** fails.
This allows the use of an input *itself* to be used as the test for failure.

Again note that if you intend to take more input from the keyboard, you must reset the keyboard with **`cin.clear`**.

Using Failed Input for Processing

Using the input attempt directly we have:

```
cout << "Enter values, Q to quit: ";
while (cin >> value)
{
   // process value here
}
cin.clear();
```

Chapter Summary

- Loops execute a block of code repeatedly while a condition remains true.
- The for loop is used when the loop body must be executed at least once.
- Nested loops are commonly used for processing tabular structures.
- A sentinel value denotes the end of data set, but it is not part of the data.
- We can use a Boolean variable to control a loop. Set the variable to true before entering the loop, then set it to false to leave the loop.