Topic 2

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

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.

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.

`while()` Loop Hand-Tracing Example

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;
```

n	sum	digit
1729	0	

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

So we make a table with a column for each variable, and fill in one iteration of the loop per row.

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

Hand-Tracing Example, continued #2

n	sum	digit
1729	0	9
	9	

Because n is greater than zero, enter the loop.

The variable digit is set to 9 (the remainder of 1729 %10).

The variable sum is set to 0 + 9 = 9, in the 2nd row of the table.

When updating variable values, cross off the obsolete values, such as sum=0 above.

Hand-Tracing Example, continued #3

Keep filling in the table as you move the clip, cycling through the loop.

After the 4th time through the loop, n=0, so the loop is not entered again…

… and execution ends with the completion of the `cout` statement printing the final sum=19.

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

Topic 3

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

The `for` Loop vs. the `while` loop

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

You could use a **`while`** loop:

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

The `for` Loop

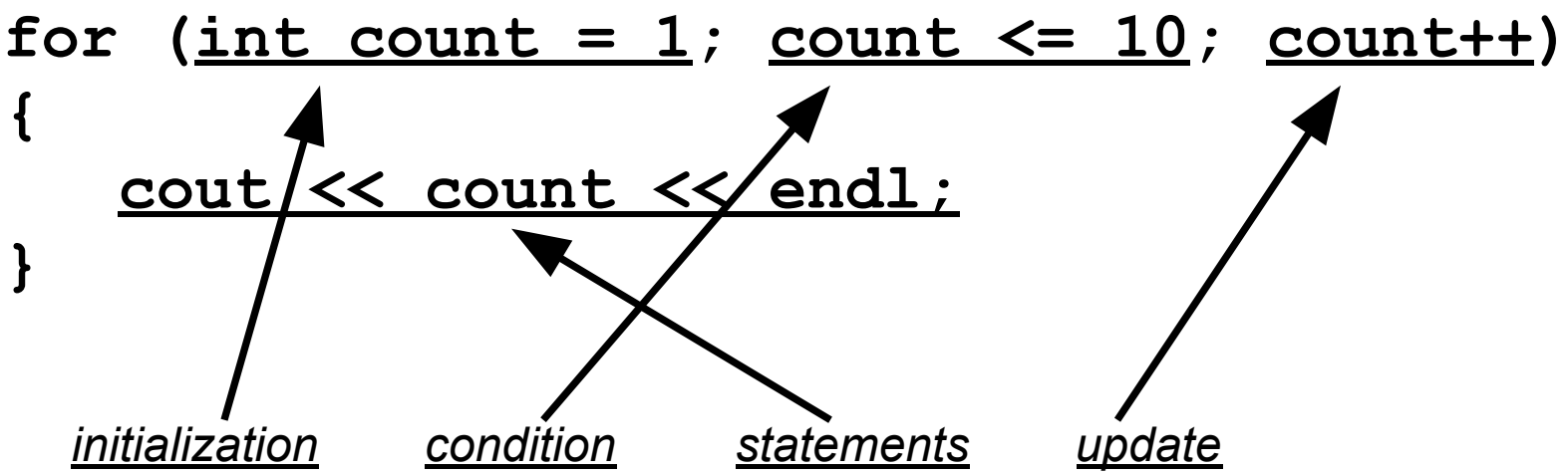
C++ has a statement custom made ***for*** this sort of processing:

the **`for`** loop.

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

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

Doing something a known 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()` loop execution

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

- The ***initialization*** is code that happens once, before the check is made, to set up counting how many times the *statements* will happen. The loop variable may be created here, or before the `for()` statement.
- The ***condition*** is a comparison to test if the loop is done. When this test is false, we skip out of the `for()`, going on to the next statement.
- The ***update*** is code that is executed at the bottom of each iteration of the loop, immediate before re-testing the condition. Usually it is a counter increment or decrement.
- The ***statements*** are repeatedly executed until the condition is false. These also are known as the "loop body".

Scope of the Loop Variable – Define it in the `for` or earlier?

- The "loop variable" can be defined inside the **`for`** parentheses:

```
int strlen_error(string s) {
   for(int i=0; s[i]!=0; i++);
  return i; //syntax error: unknown identifier "i"
}
```

  But then it 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 were previously defined
  - (In an earlier example, **`counter`** was defined before the loop – so it could be accessed after the loop exited.)

The `for` Can Count Up or Down

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

```
for (counter = 10; counter >= 0; counter--)…
```

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

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

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

`for` Loop Examples, Index Values: Table 2

Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 4.3)
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i += 2)`	0 2 4 6 8 10 ... (infinite loop)	You can use < or <= instead of != to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying i, such as doubling it in every step.
`for (i = 0; i < str.length(); i++)`	0 1 2 … until the last valid index of the string str	In the loop body, use the expression `str.substr(i, 1)` to get a `string` containing the ith character.

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`: Desired Output

The output should be a table with columns for year and balance, something like this:

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

The Modified Investment Program Using a `for` Loop

ch04/invtable.cpp

```
#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;
}
```

Infinite Loops Can Occur in `for` Statements

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

`for` loop

The output never ends

```
0 2 4 6 8 10 12...
```

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