The for Loop

Class 16

The for Loop

- C++'s third looping construct

```
for (initialization; test; update)
{
  statement;
  statement;
  ...
}
```

- example

```
for (unsigned count = 0; count < 5; count++)
{
  cout << "Hello" << endl;
}
```

The for Loop

```
for (unsigned count = 0; count < 5; count++)
{
   cout << "Hello" << endl;
}
```

- semicolons separate initialization, test, and update
- there is no semicolon after update
- the scope of a variable declared in the initialization part is inside the curly braces of the loop body block

The for Loop

```
for (unsigned count = 0; count < 5; count++)
{
  cout << "Hello" << endl;
}
```

- semicolons separate initialization, test, and update
- there is no semicolon after update
- the scope of a variable declared in the initialization part is inside the curly braces of the loop body block

- the initialization statement is only done once, the first step
- the test statement is done every time
  - right after the initialization the first time
  - right after the update on subsequent iterations
  - this makes the for loop a pretest loop construct
- the update statement is not done the first time, but is done before the test on every subsequent iteration

For Style and Philosophy

- the C++-style for loop was invented many decades ago, when programming was in its infancy
- the for loop allows you to do many things that are no longer allowed by best programming practice
- some of them generate compiler warnings; others do not
- here are some things to never do, or at least avoid unless there's a really good reason

Things to Avoid

1. declare the loop control variable before the loop header

```
unsigned count;
for (count = 0; count < 5; count++)
{ ...
```

- unless you absolutely need count to exist after the loop ends, you should declare count in the loop header like this:

```
for (unsigned count = 0; count < 5; count++)
{ ...
```

Things to Avoid

2. missing parts of the loop header

```
unsigned count = 0;
for ( ; count < 5; )
{
   ...
   count++;
}
```

- this is legal, but even worse than #1
- in old code you'll sometimes see: `for ( ; ; )` ugh!

Things to Avoid

3. multiple statements in the loop header

```
for (unsigned count = 0; count < 5; count++, foobar--)
{
   ...
```

- this is legal, but you should never do it
- the only thing the loop header should do is control the loop
- since foobar is not involved in controlling the loop, modifying it should be done in the loop body, not in the loop header

Things to Avoid

4. modifying the loop control variable in the loop body

```
for (unsigned count = 0; count < 5; count++)
{
  count += 2;
  ...
```

- never do this!
- the entire control of the loop should reside in the header

Things to Avoid

5. using a floating point value to control a for loop

```
for (double count = 0; count < 5.0; count++)
{
   ...
```

- doubles should never be used for counting purposes
- while loops, which are not conceptually count-controlled loops, can use doubles as loop control variables

Other Step Sizes

- the most common step size of for loops is positive one

```
for (unsigned count = 0; count < 5; count++)
```

- but negative one is also common

```
for (unsigned count = 5; count > 0; count--)
```

- and it's easy to count by 2's or other increments

```
for (unsigned count = 0; count < 10; count += 2)
```

Number of Loop Iterations

```
const int START_VALUE = -5;
const int STOP_VALUE = 8;
for (int control = START_VALUE; control < STOP_VALUE;
     control++)

for (int control = START_VALUE; control <= STOP_VALUE;
     control++)
```

- if a for loop acts by incrementing and the test condition is less than, the number of loop iterations will be $\texttt{STOP\_VALUE} - \texttt{START\_VALUE}$
- if a for loop acts by incrementing and the test condition is less than or equal to, the number of loop iterations will be $\texttt{STOP\_VALUE} - \texttt{START\_VALUE} + 1$

Number of Loop Iterations

```
const int START_VALUE = 8;
const int STOP_VALUE = -5;

for (int control = START_VALUE; control > STOP_VALUE;
     control--)

for (int control = START_VALUE; control >= STOP_VALUE;
     control--)
```

- if a for loop acts by decrementing and the test condition is greater than, the number of loop iterations will be START_VALUE − STOP_VALUE
- if a for loop acts by decrementing and the test condition is greater than or equal to, the number of loop iterations will be START_VALUE − STOP_VALUE + 1

Number of Loop Iterations

```
const unsigned START_VALUE = 2;
const unsigned STOP_VALUE = 21;

for (unsigned control = START_VALUE; control > STOP_VALUE;
     control += 2)
```

- if a for loop acts by incrementing or decrementing by a value different than 1
- the number of loop iterations depends on whether the step size is an even multiple of the range size
- and whether the condition includes equal to or not
- typically need to trace by hand and run some test cases to check

Infinite Loops

- be careful not to make an infinite loop!

```
for (unsigned count = 1; count != 10; count += 2)
```

- this is an infinite loop because count is going up by two on the odd numbers and will never equal 10

```
for (unsigned count = 10; count >= 0; count--)
```

- this is tricky
- this keeps going until count is negative
- but an unsigned cannot be negative — it wraps around to a huge number and keeps going
- this is an infinite loop

For Loop Accumulator

- last class we saw an example of a while loop being used to count the number of and accumulate the sum of ACT scores
- this was an appropriate use of the while loop
- it was impossible to know when the loop started how many ACT scores the user was going to enter

For Loop Accumulator

- last class we saw an example of a while loop being used to count the number of and accumulate the sum of ACT scores
- this was an appropriate use of the while loop
- it was impossible to know when the loop started how many ACT scores the user was going to enter

- a slightly different version of the same idea makes it appropriate to use a for loop
- see accumulate_for.cpp
- when the loop starts, we know exactly how many scores the user will enter

Nested Loops

- a loop that is inside another loop is called a nested loop
- see program clock.cpp

Nested Loops

- a loop that is inside another loop is called a nested loop
- see program clock.cpp

- the innermost loop will repeat 60 times for each iteration of the middle loop
- the middle loop will repeat 60 times for each iteration of the outermost loop
- the total number of times the innermost loop runs is

$$24 \times 60 \times 60 = 86{,}400$$

Nested Loops

- a loop that is inside another loop is called a nested loop
- see program clock.cpp

- the innermost loop will repeat 60 times for each iteration of the middle loop
- the middle loop will repeat 60 times for each iteration of the outermost loop
- the total number of times the innermost loop runs is

$$24 \times 60 \times 60 = 86{,}400$$

- a nested loop goes through all of its iterations each time the outer loop iterates
- the nested loop completes its iterations faster than outer loops
- the total number of iterations of a nested loop is the product of the number of iterations of the inner and outer loops

Section 5.12 Break, Continue, and Return

- your author says:

WARNING!

Use the break and continue statements with great caution. Because they bypass the normal condition that controls the loop's iterations, these statements make code difficult to understand and debug. For this reason, you should avoid using break and continue whenever possible.

- our rule is much stronger: never use break or continue!
- (break is required in a switch statement, but that's different)

Section 5.12 Break, Continue, and Return

- your author says:

WARNING!

Use the break and continue statements with great caution. Because they bypass the normal condition that controls the loop's iterations, these statements make code difficult to understand and debug. For this reason, you should avoid using break and continue whenever possible.

- our rule is much stronger: never use break or continue!
- (break is required in a switch statement, but that's different)

- we will soon talk at length about return statements, but the same logic applies:

  never return from the middle of a loop!