UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

For loops

Prof. Hiren Patel, Ph.D.
Douglas Wilhelm Harder, M.Math. LEL
hdpatel@uwaterloo.ca dwharder@uwaterloo.ca

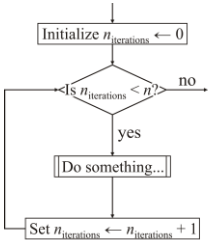
Outline

- In this lesson, we will:
  - Describe for loops and their implementation in C++
  - Describe their purpose
    - Specifically count-controlled loops

Count-controlled loops

- We previous looked at executing a block of code a fixed number of times:

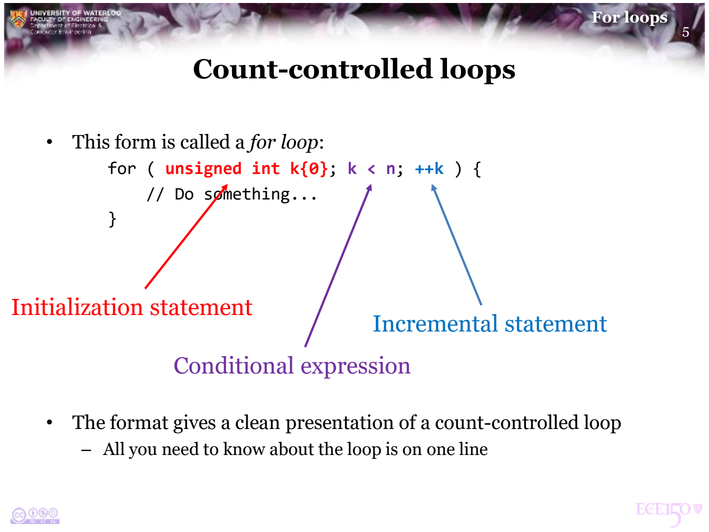
```
unsigned int num_iterations{0};

while ( num_iterations < max_iterations ) {
    // Do something...

    ++num_iterations;
}
```


Initialize $n_{\text{iterations}} \leftarrow 0$
Is $n_{\text{iterations}} < n$?
no
yes
Do something...
Set $n_{\text{iterations}} \leftarrow n_{\text{iterations}} + 1$

Count-controlled loops

- This is so common, it is given a short form:

```
unsigned int k{0};

while ( k < n ) {
    // Do something...

    ++k;
}

for ( unsigned int k{0}; k < n; ++k ) {
    // Do something...
}
```



Count-controlled loops

- This form is called a *for loop*:

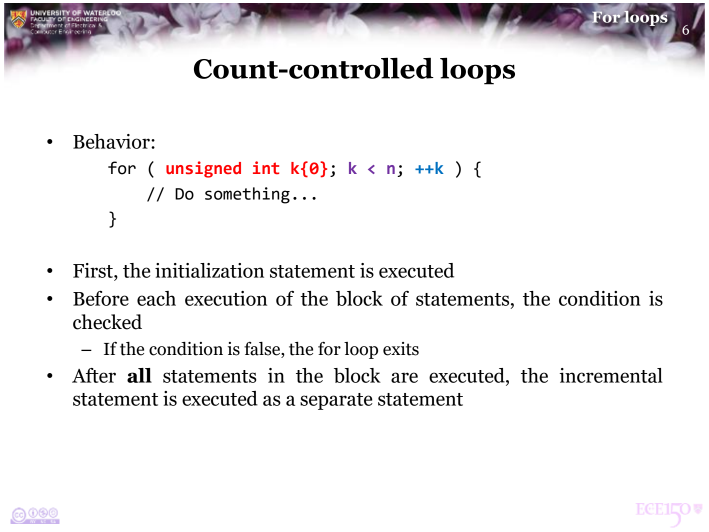
```
for ( unsigned int k{0}; k < n; ++k ) {
    // Do something...
}
```

Initialization statement

Incremental statement

Conditional expression

- The format gives a clean presentation of a count-controlled loop
  - All you need to know about the loop is on one line

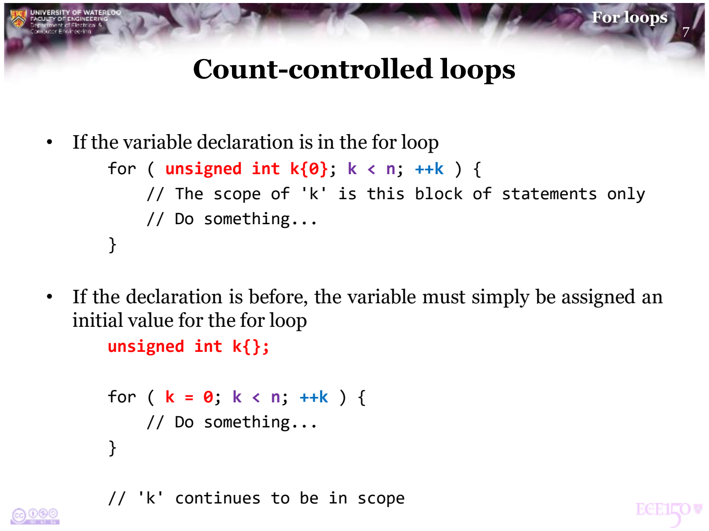


Count-controlled loops

- Behavior:

```
for ( unsigned int k{0}; k < n; ++k ) {
    // Do something...
}
```

- First, the initialization statement is executed
- Before each execution of the block of statements, the condition is checked
  - If the condition is false, the for loop exits
- After **all** statements in the block are executed, the incremental statement is executed as a separate statement



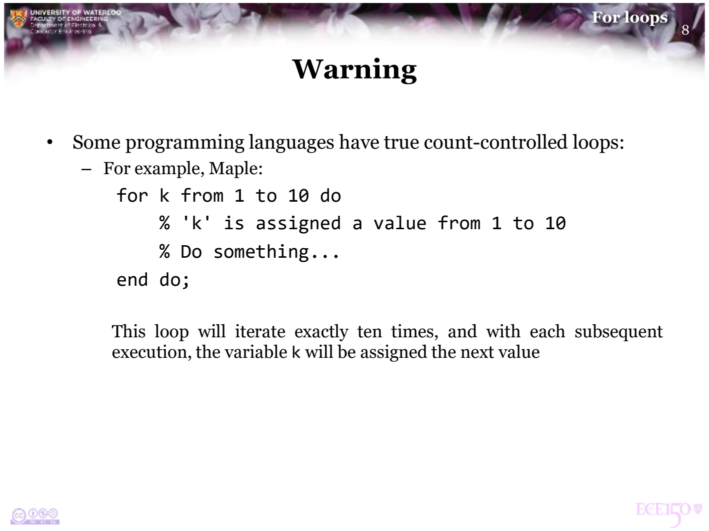
Count-controlled loops

- If the variable declaration is in the for loop

```
for ( unsigned int k{0}; k < n; ++k ) {
    // The scope of 'k' is this block of statements only
    // Do something...
}
```

- If the declaration is before, the variable must simply be assigned an initial value for the for loop

```
unsigned int k{};

for ( k = 0; k < n; ++k ) {
    // Do something...
}

// 'k' continues to be in scope
```

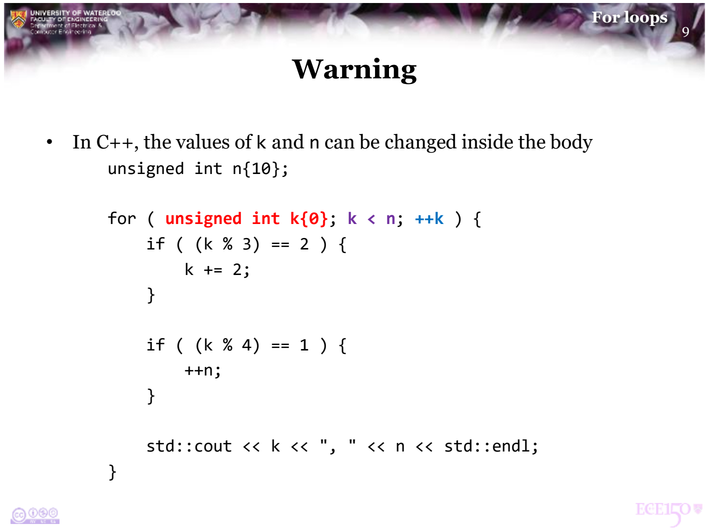


Warning

- Some programming languages have true count-controlled loops:
  - For example, Maple:

```
for k from 1 to 10 do
    % 'k' is assigned a value from 1 to 10
    % Do something...
end do;
```

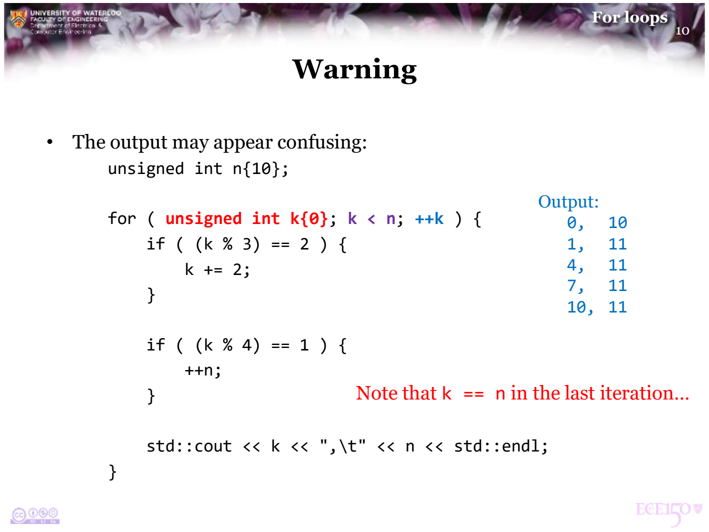
This loop will iterate exactly ten times, and with each subsequent execution, the variable k will be assigned the next value



Warning

- In C++, the values of k and n can be changed inside the body

```
unsigned int n{10};

for ( unsigned int k{0}; k < n; ++k ) {
    if ( (k % 3) == 2 ) {
        k += 2;
    }

    if ( (k % 4) == 1 ) {
        ++n;
    }

    std::cout << k << ", " << n << std::endl;
}
```



Warning

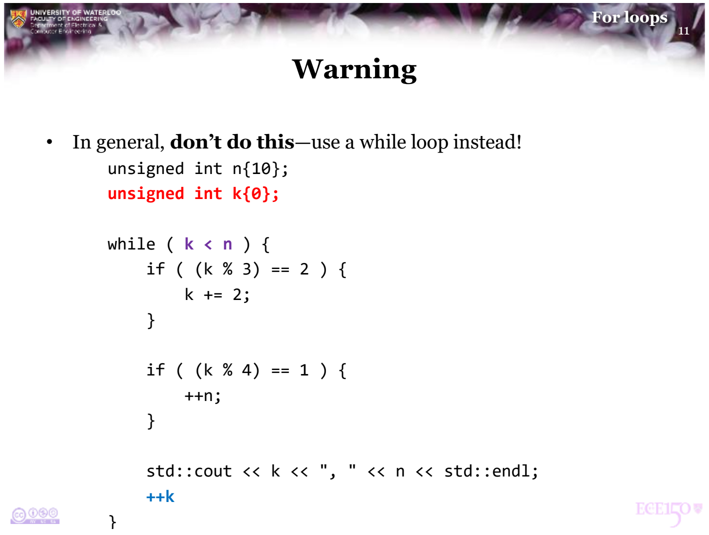
- The output may appear confusing:

```
unsigned int n{10};

for ( unsigned int k{0}; k < n; ++k ) {
    if ( (k % 3) == 2 ) {
        k += 2;
    }

    if ( (k % 4) == 1 ) {
        ++n;
    }

    std::cout << k << ",\t" << n << std::endl;
}
```

Output:

```
0,  10
1,  11
4,  11
7,  11
10, 11
```

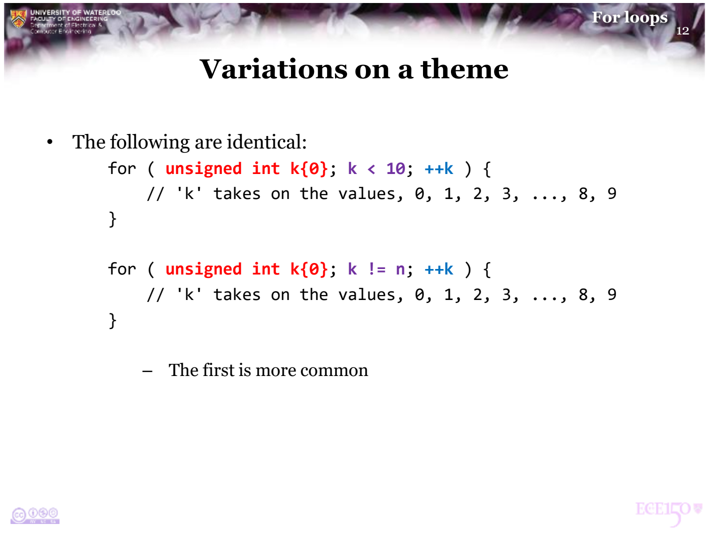
Note that k == n in the last iteration...



Warning

- In general, **don't do this**—use a while loop instead!

```
unsigned int n{10};
unsigned int k{0};

while ( k < n ) {
    if ( (k % 3) == 2 ) {
        k += 2;
    }

    if ( (k % 4) == 1 ) {
        ++n;
    }

    std::cout << k << ", " << n << std::endl;
    ++k
}
```

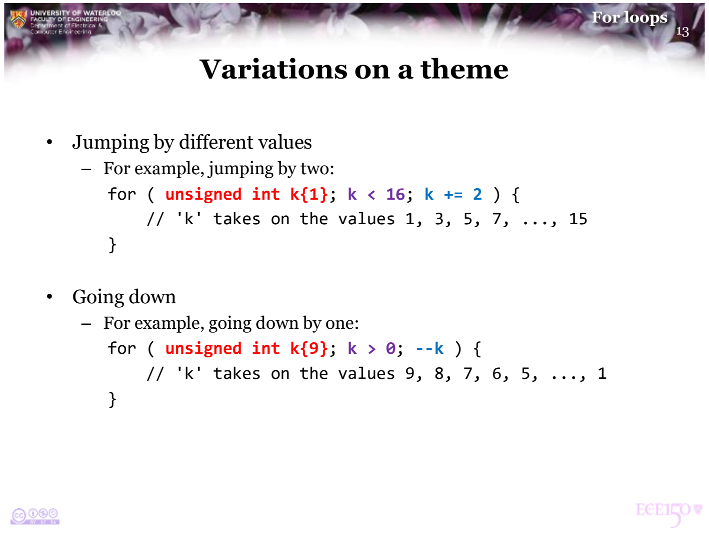


Variations on a theme

- The following are identical:

```
for ( unsigned int k{0}; k < 10; ++k ) {
    // 'k' takes on the values, 0, 1, 2, 3, ..., 8, 9
}

for ( unsigned int k{0}; k != n; ++k ) {
    // 'k' takes on the values, 0, 1, 2, 3, ..., 8, 9
}
```

  - The first is more common



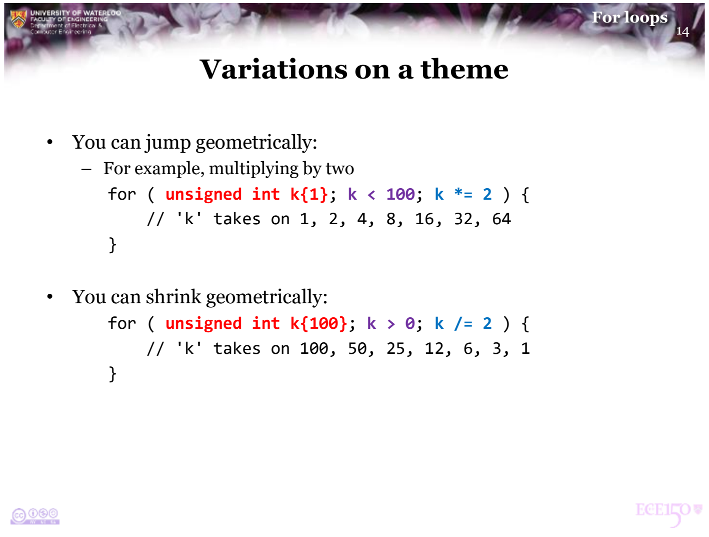
Variations on a theme

- Jumping by different values
  - For example, jumping by two:

```
for ( unsigned int k{1}; k < 16; k += 2 ) {
    // 'k' takes on the values 1, 3, 5, 7, ..., 15
}
```

- Going down
  - For example, going down by one:

```
for ( unsigned int k{9}; k > 0; --k ) {
    // 'k' takes on the values 9, 8, 7, 6, 5, ..., 1
}
```



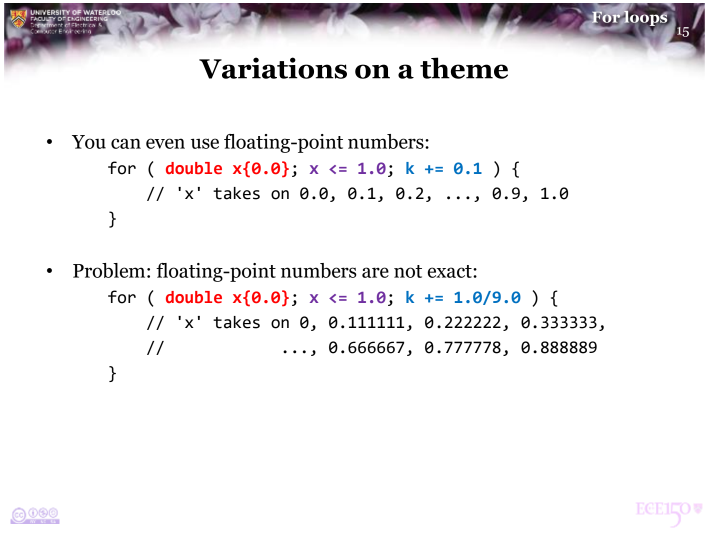
Variations on a theme

- You can jump geometrically:
  - For example, multiplying by two

```
for ( unsigned int k{1}; k < 100; k *= 2 ) {
    // 'k' takes on 1, 2, 4, 8, 16, 32, 64
}
```

- You can shrink geometrically:

```
for ( unsigned int k{100}; k > 0; k /= 2 ) {
    // 'k' takes on 100, 50, 25, 12, 6, 3, 1
}
```



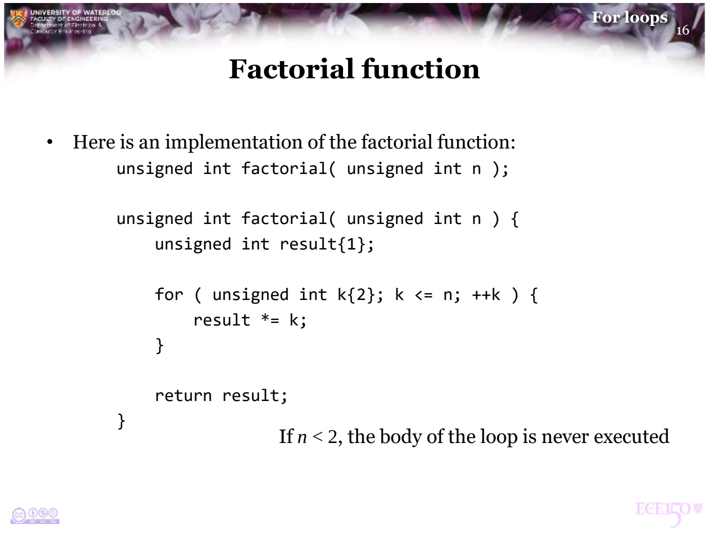
Variations on a theme

- You can even use floating-point numbers:

```
for ( double x{0.0}; x <= 1.0; k += 0.1 ) {
    // 'x' takes on 0.0, 0.1, 0.2, ..., 0.9, 1.0
}
```

- Problem: floating-point numbers are not exact:

```
for ( double x{0.0}; x <= 1.0; k += 1.0/9.0 ) {
    // 'x' takes on 0, 0.111111, 0.222222, 0.333333,
    //            ..., 0.666667, 0.777778, 0.888889
}
```



Factorial function

- Here is an implementation of the factorial function:

```
unsigned int factorial( unsigned int n );

unsigned int factorial( unsigned int n ) {
    unsigned int result{1};

    for ( unsigned int k{2}; k <= n; ++k ) {
        result *= k;
    }

    return result;
}
```

If $n < 2$, the body of the loop is never executed

Factorial function

- Try this yourself:

```
#include <iostream>

// Function declarations
int main();
unsigned int factorial( unsigned int n );

// Function definitions
int main() {
    for ( int k{0}; k < 20; ++k ) {
        std::cout << k << "! = " << factorial( k ) << std::endl;
    }

    return 0;
}
unsigned int factorial( unsigned int n ) {
    unsigned int result{1};

    for ( unsigned int k{2}; k <= n; ++k ) {
        result *= k;
    }

    return result;
}
```

Perfect numbers

- A number is *perfect*—whatever that means—if it is the sum of its divisors

```
bool is_perfect( unsigned int n );

bool is_perfect( unsigned int n ) {
    unsigned int sum{0};

    for ( unsigned int k{1}; k < n; ++k ) {
        if ( (n % k) == 0 ) {
            sum += k;
        }
    }

    return (sum == n);
}
```

Prime numbers

- A number *n* is *prime* if it is not divisible by any number between 2 and *n* – 1:

```
bool is_prime( unsigned int n );

bool is_prime( unsigned int n ) {
    for ( unsigned int k{2}; k < n; ++k ) {
        if ( (n % k) == 0 ) {
            return false;
        }
    }

    return true;
}
```

Prime numbers

- You really only need to search up to the integer square root of *n*:

```
bool is_prime( unsigned int n );

bool is_prime( unsigned int n ) {
    unsigned int upper_bound{isqrt(n)};

    for ( unsigned int k{2}; k <= upper_bound; ++k ) {
        if ( (n % k) == 0 ) {
            return false;
        }
    }

    return true;
}
```

Summary

- Following this lesson, you now
  - Understand how to implement for loops in C++
  - Know this is a special case of the while loop
  - Understand it should be restricted to count-controlled loops
  - Seen various applications

References

[1] Wikipedia
https://en.wikipedia.org/wiki/For_loop

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.