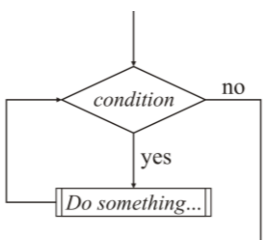
Outline

- In this lesson, we will:
  - Describe looping statements and their implementation in C++
  - Introduce non-terminating loops
  - Implement two versions examining the Collatz conjecture
  - See how to limit the number of iterations of a loop
  - Implement the factorial function
  - Walk through the steps of converting an algorithm described in English to a program
    - We will use the greatest-common divisor algorithm

Looping statements

- We previously looked at:
  - Executing blocks of statements
  - Conditionally executing a block of statements based on a Boolean-valued condition
- We will now look at the C++ implementation of a looping statement



Looping statements

- We implemented conditional statements based on a condition being evaluated to TRUE

```
if ( condition ) {
    // Do something...
} else {
    // Do something else...
}
```

Looping statements

- A looping statement is implemented based on repeatedly executing a block of statements so long as a condition is TRUE

```
while ( Boolean-valued condition ) {
    The looped block of statements
     - to be executed as long as the
       condition is 'true'
}

// Continue executing here as soon as the
// condition evaluates to 'false'
```


Looping statements

- Because this state block is repeatedly executed (i.e., looped) while a condition is TRUE, we refer to such a statement as a “while loop”

```
while ( Boolean-valued condition ) {
    The looped block of statements
     - to be executed as long as the
       condition is 'true'
}

// Continue executing here as soon as the
// condition evaluates to 'false'
```


Looping statements

- The easiest while loop is one that does so forever:

```
#include <iostream>

int main();

int main() {
    // @non-terminating@
    while ( true ) {
        std::cout << "Hello world!" << std::endl;
    }

    return 0;
}
```


Looping statements

- Such non-terminating while loops are used in real-time systems that respond to external events:

```
int main();

int main() {
    // @non-terminating@
    while ( true ) {
        // Wait for an event
        // Respond to that event
    }

    // Technically, we never get here
    return 0;
}
```

Collatz conjecture

- Normally, however, the condition is affected by the action of the looped block of statements
- We'll look at one example based on an interesting mathematical quandary:
  - The Collatz conjecture says that if you start with a number a, do the following:
    - If it is odd, multiply it by three and add one
    - If it is even, divide it by two
  - The Collatz conjecture says that this sequence will ultimately reduce to the cycle 1, 4, 2, 1, 4, 2, 1, …

Collatz conjecture

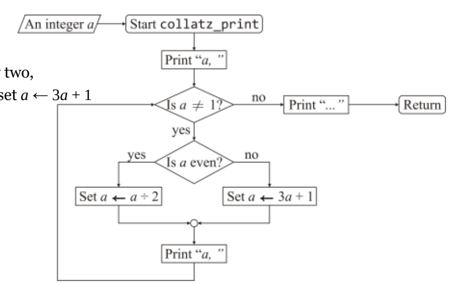
- We can try this with any number of initial values

  1
  2, 1
  3, 10, 5, 16, 8, 4, 2, 1
  4, 2, 1
  5, 16, 8, 4, 2, 1
  6, 3, 10, 5, 16, 8, 4, 2, 1
  7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
  8, 4, 2, 1
  9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
  10, 5, 16, 8, 4, 2, 1
  11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
  12, 6, 3, 10, 5, 16, 8, 4, 2, 1

Collatz conjecture

- Write a function `collatz_print(…)` that takes a positive integer a and prints out the sequence of integers until it reaches one, in which case, terminate with `"..."`

1. Print "a, "
2. If $a \neq 1$,
   a. If a is even,
      i. Divide a by two,
      ii. Otherwise, set $a \leftarrow 3a + 1$
   b. Print "a, "
   c. Go to Step 2.
3. Print "…"



Collatz conjecture

- An implementation of this algorithm is:

```
void collatz_print( unsigned int a );

void collatz_print( unsigned int a ) {
    std::cout << a << ", ";

    while ( a != 1 ) {
        if ( (a % 2) == 0 ) {
            a /= 2;
        } else {
            a = 3*a + 1;
        }

        std::cout << a << ", ";
    }

    std::cout << "..." << std::endl;
}
```

Question: What happens if the argument passed is `0`?

Collatz conjecture

- Try it yourself:

```
#include <iostream>
#include <cassert>

// Function declarations
void collatz_print( unsigned int a );

// Function definitions
void collatz_print( unsigned int a ) {
    assert( a != 0 );
    std::cout << a << ", ";

    while ( a != 1 ) {
        if ( (a % 2) == 0 ) {
            a /= 2;
        } else {
            a = 3*a + 1;
        }

        std::cout << a << ", ";
    }

    std::cout << "..." << std::endl;
}

int main() {
    collatz_print( 1970 );
    return 0;
}
```


Collatz conjecture

- Mathematicians aren't so interested in the actual sequences, but rather the number of terms in the sequence until you get to 1
  - For example,

112 iterations

27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

125 iterations

171, 514, 257, 772, 386, 193, 580, 290, 145, 436, 218, 109, 328, 164, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Collatz conjecture

- Write a function `collatz(…)` that takes a positive integer *n* and returns the number of steps required until we get to one:

```
unsigned int collatz( unsigned int a );

unsigned int collatz( unsigned int a ) {
    unsigned int num_iterations{0};

    while ( a != 1 ) {
        ++num_iterations;

        if ( (a % 2) == 0 ) {
            a /= 2;
        } else {
            a = 3*a + 1;
        }
    }

    return num_iterations;
}
```


Collatz conjecture

- Notice the function declarations:

```
void collatz_print( unsigned int n );
unsigned int collatz( unsigned int n );
```

- Could we not just give the same name?
  - After all, we had other functions with the same name
- Problem: The C++ compiler **cannot** choose based on return type only
  - The compiler only chooses based on the types of any arguments
  - If two functions have identical parameter types, they must have different names

Collatz conjecture

- We could create a second loop that queries the user for an argument:

```
#include <iostream>
int main();
void collatz_print( unsigned int n );

int main() {
    bool keep_going{true};

    while ( keep_going ) {
        unsigned int n{};
        std::cout << "Enter a positive integer ('0' to quit): ";
        std::cin >> n;

        if ( n == 0 ) {
            keep_going = false;
        } else {
            collatz_print( n );
        }
    }

    return 0;
}
```


Collatz conjecture

- Problem: what happens if we pass our function the argument `0`?
  - While the specification may require the argument to be greater than zero, you must explicitly check:

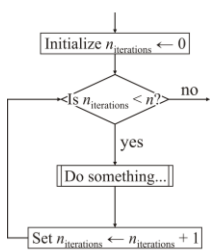
```
unsigned int collatz( unsigned int n ) {
    assert( n >= 1 );

    unsigned int num_iterations{0};
    // Other code...
    return num_iterations;
}
```

```
unsigned int collatz( unsigned int n ) {
    if ( n == 0 ) {
        return 0;
    } else {
        unsigned int num_iterations{0};
        // Other code...
        return num_iterations;
    }
}
```


Counting

- Suppose we want a loop to run exactly n times
  - Track of how often the loop has executed with a local variable $n_{\text{iterations}}$
  1. Initialize a local variable $n_{\text{iterations}} \leftarrow 0$
  2. As long as $n_{\text{iterations}} < n$,
     a. execute the block of statements associated with the loop,
     b. increment $n_{\text{iterations}}$, and
     c. return to Step 2.



Counting

- Here is a while loop that executes a fixed number of times
  - This assumes the value of $max_{\text{iterations}}$ is never changed...

```
unsigned int num_iterations{0};

while ( num_iterations < max_iterations ) {
    // Do something...
    ++num_iterations;
}
```

- Once `num_iterations == max_iterations`, we have executed the block of statements the required number of times

Factorial function

- Suppose we want to calculate $n!$
  – Because $0! = 1! = 1$, we could start with this value, and then keep multiplying this by 2, then 3, and so on up until n:
  1. Initialize a result $r \leftarrow 1$ and a variable $k \leftarrow 2$
  2. If $k \leq n$,
     a. multiply r by k: $r \leftarrow kr$,
     b. increment $k \leftarrow k + 1$, and
     c. return to Step 2.
  3. Return r

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};
    unsigned int k{2};

    while ( k <= n ) {
        result *= k;
        ++k;
    }

    return result;
}
```


How to design a while loop

- Suppose you are attempting to implement an algorithm where you repeated apply a number of steps
  – How do you make the transition from manual to programmatic?

- Recommendation:
  – Do the algorithm on paper—in full
  – Examine the steps you took, and determine:
    - What steps were repeated?
    - What condition caused you to stop repeating the steps?

The greatest-common divisor

- From secondary school, you saw that the algorithm for calculating the greatest common denominator (gcd)
  – You are asked to find the gcd of 8008 and 8085
  – You first note that 8085 > 8008
  – Next, you find that 8085 ÷ 8008 equals 1 with a remainder of 77
  – Next, you find that 8008 ÷ 77 equals 104 with a remainder of 0
  – From this, you are told that the gcd is 77

The greatest-common divisor

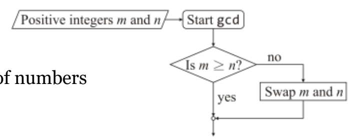
- Let's try again:
  - You are asked to find the gcd of 1583890 and 85800
  - You first note that 1583890 > 85800
  - Next, you find that 1583890 ÷ 85800 equals 18 with a remainder of 39490
  - Next, you find that 85800 ÷ 39490 equals 2 with a remainder of 6820
  - Next, you find that 39490 ÷ 6820 equals 5 with a remainder of 5390
  - Next, you find that 6820 ÷ 5390 equals 1 with a remainder of 1430
  - Next, you find that 5390 ÷ 1430 equals 3 with a remainder of 1100
  - Next, you find that 1430 ÷ 1100 equals 1 with a remainder of 330
  - Next, you find that 1100 ÷ 330 equals 3 with a remainder of 110
  - Next, you find that 330 ÷ 110 equals 3 with a remainder of 0
  - From this, you are told that the gcd is 110

The greatest-common divisor

- At each step, we had a pair of numbers
  - Call the *m* and *n*
- For the initial step, we set
  - the larger number to be *m*, and
  - the smaller number to be *n*
- After that, we repeatedly
  - calculated the remainder *r* when dividing $m \div n$, and
  - if $r = 0$, we are done, and *n* is the gcd,
  - otherwise, we repeat with the pair *n* and *r*
    - That is, we set $m \leftarrow n$ and then we set $n \leftarrow r$

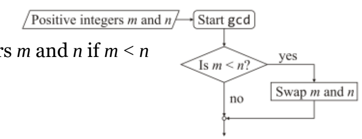
Remember: when you calculate $m \div n$,
the remainder *r* must always satisfy $n > r \geq 0$

The greatest-common divisor



- Let's look at the first steps:
  - At each step, we had a pair of numbers
    - Call them *m* and *n*
  - For the initial step, we set
    - the larger number to be *m*, and
    - the smaller number to be *n*
- Problem: We are doing something if the condition is false:
  - We execute a block of statements when $m \geq n$ is FALSE
- Let's use the complementary condition:
  - This is equivalent to executing the statements when $m < n$ is TRUE

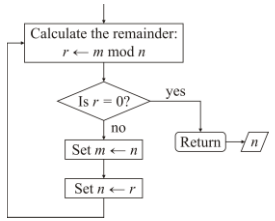
The greatest-common divisor



- Thus, we swap the parameters *m* and *n* if $m < n$

The greatest-common divisor

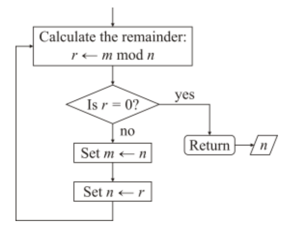
- The next steps we perform are that we
  - calculate the remainder r, and
  - if $r = 0$, we are done, and n is the gcd,
  - otherwise, we repeat with the pair n and r
    - That is, we set $m \leftarrow n$ and then we set $n \leftarrow r$

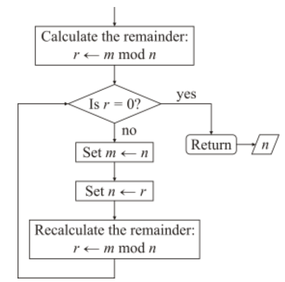


- Problem: this is not in the form a *while* loop

The greatest-common divisor

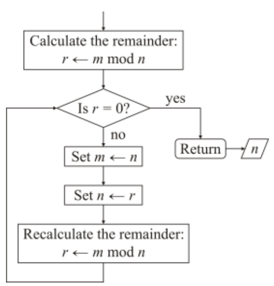
- First, a while loop must return to the condition
  - We can repeat the first statement at the end

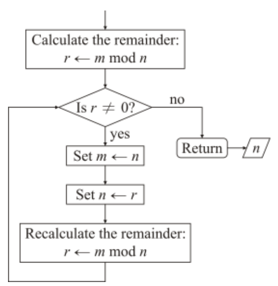




The greatest-common divisor

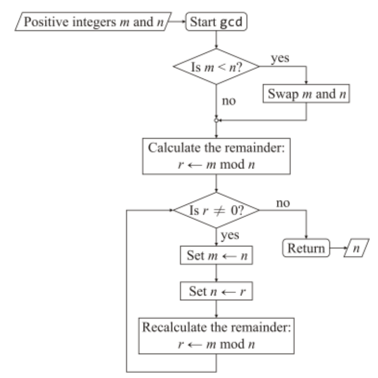
- Second, the condition for looping must evaluate to TRUE
  - Continuing while $r = 0$ is FALSE is equivalent to continuing while $r \neq 0$ is TRUE





The greatest-common divisor

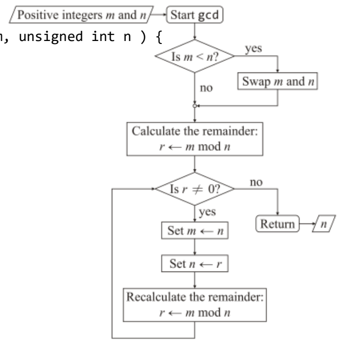
- Thus, our final flow chart is:



The greatest-common divisor

- Programming this:

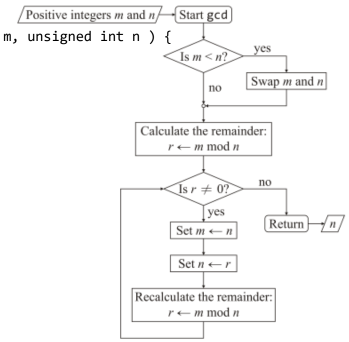
```
unsigned int gcd( unsigned int m, unsigned int n ) {
```



The greatest-common divisor

- Programming this:

```
unsigned int gcd( unsigned int m, unsigned int n ) {
    if ( m < n ) {
        unsigned int tmp{m};
        m = n;
        n = tmp;
    }
```

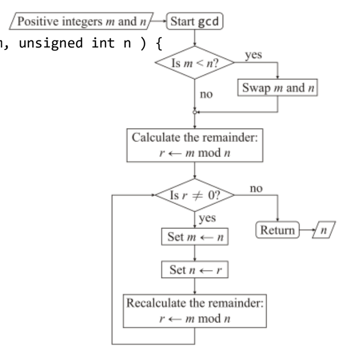


The greatest-common divisor

- Programming this:

```
unsigned int gcd( unsigned int m, unsigned int n ) {
    if ( m < n ) {
        unsigned int tmp{m};
        m = n;
        n = tmp;
    }

    unsigned int r{m % n};
```



The greatest-common divisor

- Programming this:

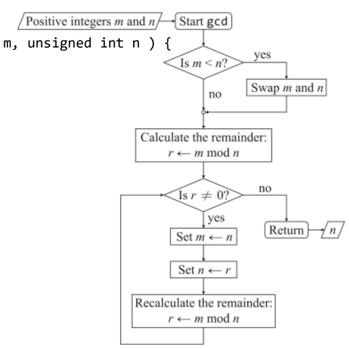
```
unsigned int gcd( unsigned int m, unsigned int n ) {
    if ( m < n ) {
        unsigned int tmp{m};
        m = n;
        n = tmp;
    }

    unsigned int r{m % n};

    while ( r != 0 ) {



    }
```



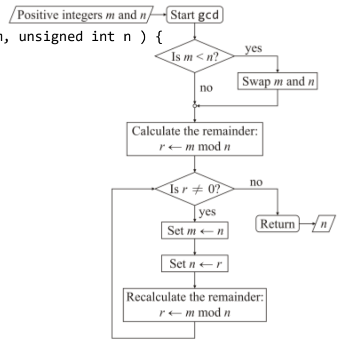
The greatest-common divisor

- Programming this:

```
unsigned int gcd( unsigned int m, unsigned int n ) {
    if ( m < n ) {
        unsigned int tmp{m};
        m = n;
        n = tmp;
    }

    unsigned int r{m % n};

    while ( r != 0 ) {
        m = n;
        n = r;
        r = m % n;
    }
```



The greatest-common divisor

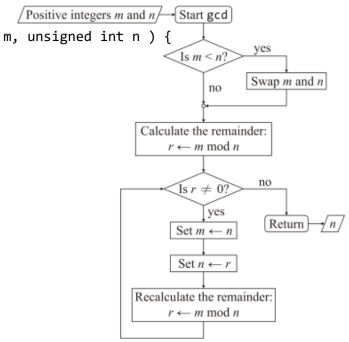
- Programming this:

```
unsigned int gcd( unsigned int m, unsigned int n ) {
    if ( m < n ) {
        unsigned int tmp{m};
        m = n;
        n = tmp;
    }

    unsigned int r{m % n};

    while ( r != 0 ) {
        m = n;
        n = r;
        r = m % n;
    }

    return n;
}
```



Infinite loop?

- Question:
  - What do you do if you accidentally execute a program that has an infinite loop?
- Solution:
  - In Eclipse, there is a *stop* button that becomes active when a program is executing

  - Other ides will have similar features
  - At the console, press Ctrl-C

Summary

- Following this lesson, you now
  - Understand how to implement while loops in C++
  - Understand how to limit the number of times the corresponding statement block is executed
  - Seen how to implement various functions requiring looping statements:
    - The Collatz conjecture
    - The factorial function
  - Understand how to convert a description of an algorithm to one that you can program
    - The example we used was the greatest-common divisor
  - Know how to terminate a program in an infinite loop

References

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

[2] cplusplus.com
http://www.cplusplus.com/doc/tutorial/control/

[3] tutorialspoint
https://www.tutorialspoint.com/cplusplus/cpp_while_loop.htm

Acknowledgments

Proof read by Dr. Thomas McConkey and Charlie Liu.

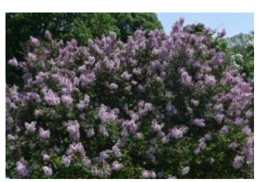
Colophon

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