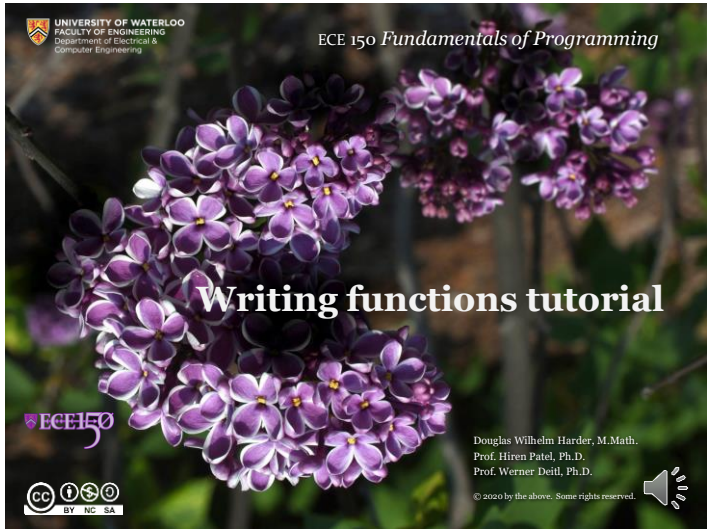

Outline

- In this tutorial, we will:
  - Describe a function for you to author
  - Have you author that function
  - Look at one possible solution and test file

Tutorial format

- In this tutorial, you will author a number of functions
  - You will always be required to author a program to test your function or functions
  - The format will be:

```
#include <iostream>
// ...other include directives

// Function declarations
int main();
// ...other function declarations

// Function definitions
int main() {
    // A program to test your function...
    return 0;
}

// ...other function definitions
```


Tutorial format

- In each case, you will be given the requirements of the function and the opportunity to implement these on your own
- We will then provide you with a solution and a discussion
  - Looking at our solution before you try authoring it yourself will not help you
  - Looking at the solutions will not help you when you must author functions on any examination

Accessing a number greater than or equal to a given number

- Author a function that takes two arguments:
  - A string that will be displayed to the user
  - An integer `lower_bound`
- The function will repeatedly query the user by printing the string and retrieving a value form console input until the user has entered a number greater than or equal to the lower bound
  - When the user has entered such a number, that value is returned
- The function declaration will be

```
int get_value_no_less_than( std::string text, int lower_bound );
```

- You must include the string library:

```
#include <string>
```


Accessing a number greater than or equal to a given number

- A solution:

```
int get_value_no_less_than( std::string text, int lower_bound ) {
    while ( true ) {
        int value{};
        std::cout << text;
        std::cin >> value;

        if ( value >= lower_bound ) {
            return value;
        }
    }
}
```


Accessing a number greater than or equal to a given number

```
#include <iostream>
#include <string>

// Function declarations
int main();
int get_value_no_less_than( std::string text, int lower_bound );

// Function definitions
int main() {
    int n{ get_value_no_less_than( "Enter a positive integer: ", 1 ) };
    std::cout << "The user entered " << n << std::endl;

    n = get_value_no_less_than( "Enter a non-negative integer: ", 0 );
    std::cout << "The user entered " << n << std::endl;

    n = get_value_no_less_than(
           "Enter an integer greater than 100: ", 101 );
    std::cout << "The user entered " << n << std::endl;

    return 0;
}
// ...other function definitions
```


Accessing a number greater than or equal to a given number

- At this point, if necessary, you should be able to create a function that requires the user to:
  - Enter a number no greater than a given number
  - Enter a number that falls between two given numbers
- You should be able to do this for both type `int` and type `double`

Binomial coefficients

- Author a function that takes two arguments:
  - Two integers n and k
- If $n < 0$, return 0
- If $k < 0$ or $k > n$, return 0
- Otherwise, return

$$\binom{n}{k} \stackrel{\text{def}}{=} \frac{n!}{k!(n-k)!}$$

- The function declaration will be

```
int binomial( int n, int k );
```


Binomial coefficients

- A solution:

```
int factorial( int n ) {
    if ( n < 0 ) {
        return 0;
    } else {
        int result{1};

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

        return result;
    }
}
```

```
int binomial( int n, int k ) {
    return factorial( n )/factorial( k )/factorial( n - k );
}
```


Binomial coefficients

```
#include <iostream>

// Function declarations
int main();
int factorial( int n );
int binomial( int n, int k );

// Function definitions
int main() {
    for ( int n{0}; n <= 5; ++n ) {
        for ( int k{-1}; k <= (n + 1); ++k ) {
            std::cout << binomial( n, k ) << " ";
        }

        std::cout << std::endl;
    }

    return 0;
}
// ...other function definitions
```

Expected output:

```
0 1 0
0 1 1 0
0 1 2 1 0
0 1 3 3 1 0
0 1 4 6 4 1 0
0 1 5 10 10 5 1 0
```


Falling factorial function

- This may seem a little odd, but related to factorials are Pochhammer symbols where

$$(n)_k = n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$$

- For example,

$$(5)_3 = 5 \cdot 4 \cdot 3 = 60 \quad \text{and} \quad (7)_4 = 7 \cdot 6 \cdot 5 \cdot 4 = 840$$

- Author a function

```
int falling_factorial( int n, int k );
```

that makes this calculation
  - If $k = 0$, return 1
  - If $k < 0$, just return 0
- When you are finished, write or rewrite the factorial function to calculate $n!$ using your new falling factorial function

Falling factorial function

- A solution:

```
int falling_factorial( int n, int k ) {
    if ( k < 0 ) {
        return 0;
    } else {
        int result{1};

        for ( int i{n}; i >= n - k + 1; --i ) {
            result *= i;
        }

        return result;
    }
}

int factorial( int n ) {
    return falling_factorial( n, n );
}
```

$$n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$$

Falling factorial function

```
#include <iostream>

// Function declarations
int main();
int factorial( int n );
int falling_factorial( int n, int k );

// Function definitions
int main() {
    std::cout << falling_factorial( 5, 0 ) << " = 1" << std::endl;
    std::cout << falling_factorial( 5, 1 ) << " = 5" << std::endl;
    std::cout << falling_factorial( 5, 2 ) << " = 20" << std::endl;
    std::cout << falling_factorial( 5, 3 ) << " = 60" << std::endl;
    std::cout << falling_factorial( 5, 4 ) << " = 120" << std::endl;
    std::cout << falling_factorial( 5, 5 ) << " = 120" << std::endl;
    std::cout << falling_factorial( 5, 6 ) << " = 0" << std::endl;

    return 0;
}

// ...other function definitions
```

Expected output:

```
1 = 1
5 = 5
20 = 20
60 = 60
120 = 120
0 = 0
```


Binomial coefficients

- In your previous function, you may have been a little more clever in implementing an algorithm for calculating the binomial coefficients
  - We can simplify this as follows:

$$\binom{n}{k} \stackrel{\text{def}}{=} \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot (n-1) \cdots (n-k+1)}{k!}$$

  - This reduces the amount of work required
    - For example, to calculate `binomial( 10, 4 )`, using the previous algorithm, we had to calculate

$$\frac{10 \cdot 9 \cdot 8 \cdot 7 \cdot \cancel{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}}{(4 \cdot 3 \cdot 2 \cdot 1)\cancel{(6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1)}}$$

    - Instead, there is much less work required if we calculate:

$$\frac{10 \cdot 9 \cdot 8 \cdot 7}{4 \cdot 3 \cdot 2 \cdot 1}$$

Binomial coefficients

- A solution:

```
int binomial( int n, int k ) {
    return falling_factorial( n, k )/factorial( k );
}
```

Binomial coefficients

- We will make one final change to our `binomial(…)` function:

$$\binom{10}{8} = \frac{10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3}{8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}$$

- However,

$$\binom{10}{8} = \binom{10}{2} = \frac{10 \cdot 9}{2 \cdot 1}$$

- Which is easier to calculate?
  - Implement this change in your binomial function

Binomial coefficients

- A solution:

```
int binomial( int n, int k ) {
    if ( 2*k > n ) {
        k = n - k;
    }

    return falling_factorial( n, k )/factorial( k );
}
```

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
```

- An alternative solution:

```
int binomial( int n, int k ) {
    if ( 2*k <= n ) {
        return falling_factorial( n, k )/factorial( k );
    } else {
        return falling_factorial( n, n - k )/factorial( n - k );
    }
}
```


Arithmetic series

- An arithmetic series is the sum:

$$a + (a+d) + (a+2d) + (a+3d) + \cdots + (a+(n-1)d) + (a+nd)$$

- For example,

$$3.1 + 5.1 + 7.1 + 9.1 + 11.1 = 35.5$$

- Write a function

```
double arithmetic( double a, double d, int n );
```

- If $n < 0$, return 0

Arithmetic series

- A solution:

```
double arithmetic( double a, double d, int n ) {
    if ( n < 0 ) {
        return 0.0;
    } else {
        double sum{0.0};

        for ( int k{0}; k <= n; ++k ) {
            sum += a + k*d;
        }

        return sum;
    }
}
```

Arithmetic series

```
#include <iostream>

// Function declarations
int main();
double arithmetic( double a, double d, int n );

// Function definitions
int main() {
    std::cout << arithmetic( 1.5, 0.8, 6 ) << " = 27.3" << std::endl;
    std::cout << arithmetic( 7.8, -1.2, 10 ) << " = 19.8" << std::endl;
    std::cout << arithmetic( 2.4, 0.0, 10 ) << " = " << (11*2.4)
              << std::endl;
    std::cout << arithmetic( 2.4, 9.5, 0 ) << " = 2.4" << std::endl;
    std::cout << arithmetic( 2.4, 9.5, -1 ) << " = 0" << std::endl;

    return 0;
}

// ...other function definitions
```

Expected output:

```
27.3 = 27.3
19.8 = 19.8
26.4 = 26.4
2.4 = 2.4
0 = 0
```


Arithmetic series

- Note that we can rewrite an arithmetic series as:

$$a+(a+d)+(a+2d)+(a+3d)+\cdots+(a+(n-1)d)+(a+nd)$$
$$=(a+0d)+(a+1d)+(a+2d)+(a+3d)+\cdots+(a+(n-1)d)+(a+nd)$$
$$=(n+1)a+d(0+1+2+3+\cdots+(n-1)+n)$$

- However, from secondary school, you know that

$$0+1+2+3+\cdots+(n-1)+n=\frac{n(n+1)}{2}$$

- Therefore,

$$a+(a+d)+(a+2d)+(a+3d)+\cdots+(a+(n-1)d)+(a+nd)$$
$$=a(n+1)+d\frac{n(n+1)}{2}$$

Arithmetic series

- A solution:

```
double arithmetic( double a, double d, int n ) {
    if ( n < 0 ) {
        return 0.0;
    } else {
        return a*(n + 1) + d*( (n*(n + 1))/2 );
    }
}
```

- Why is this factoring out an $(n + 1)$ a mistake?

```
double arithmetic( double a, double d, int n ) {
    if ( n < 0 ) {
        return 0.0;
    } else {
        return (n + 1)*(a + d*(n/2));
    }
}
```

```
return (n + 1)*(a + d*(n/2.0));
return (n + 1)*(a + (d*n)/2);
```


Geometric series

- Next, look online for a formula for the geometric series:

$$a+ar+ar^2+ar^3+\cdots+ar^{n-1}+ar^n$$

- Implement this as a function

```
double geometric( double a, double r, int n );
```

- You will have to include the C math library and use the `pow(…)` function

```
#define _USE_MATH_DEFINES
#include <cmath>
```

- Again, if $n < 0$, return 0

Geometric series

- A solution:

```
double geometric( double a, double r, int n ) {
    if ( n < 0 ) {
        return 0.0;
    } else {
        return a*(1.0 - std::pow( r, n + 1 ))/(1.0 - r);
    }
}
```

$$a\frac{1-r^{n+1}}{1-r}$$

Geometric series

```
#include <iostream>

// Function declarations
int main();
double arithmetic( double a, double d, int n );

// Function definitions
int main() {
    std::cout << geometric( 7.8, 1.1, 4 ) << " = 47.61978" << std::endl;
    std::cout << geometric( 5.3, -1.5, 5 ) << " = -22.028125"
              << std::endl;
    std::cout << geometric( 7.8, 1.0, 10 ) << " = " << (11*7.8)
              << std::endl;
    std::cout << geometric( 2.4, 9.5, 0 ) << " = 2.4" << std::endl;
    std::cout << geometric( 2.4, 9.5, -1 ) << " = 0" << std::endl;

    return 0;
}

// ...other function definitions
```

Expected output:

```
47.6198 = 47.61978
-22.0281 = -22.028125
85.8 = 85.8
2.4 = 2.4
0 = 0
```


Geometric series

- A solution:

```
double geometric( double a, double r, int n ) {
    if ( n < 0 ) {
        return 0.0;
    } else {
        return a*(1.0 - std::pow( r, n + 1 ))/(1.0 - r);
    }
}
```

$$a\frac{1-r^{n+1}}{1-r}$$

Geometric series

- A solution:

```
double geometric( double a, double r, int n ) {
    if ( n < 0 ) {
        return 0.0;
    } else if ( r == 1.0 ) {
        return a*(n + 1);
    } else {
        return a*(1.0 - std::pow( r, n + 1 ))/(1.0 - r);
    }
}
```

Running averages

- Write a function that
  - Repeatedly:
    - Asks the user for a real number
    - Prints out the average of all the numbers entered so far
    - If the most recent number was zero,
      return the average of all the numbers
- Your main function should simply call this function:

```
int main() {
    std::cout << running_average() << std::endl;

    return 0;
}
```


Hint

- As the user enters more and more numbers, the values returned are:

$$\frac{a_1}{1}, \frac{a_1+a_2}{2}, \frac{a_1+a_2+a_3}{3}, \frac{a_1+a_2+a_3+a_4}{4}, \ldots$$

- Note that we only have to keep a sum that equals all values currently entered so far:
  - If we have the sum $a_1 + a_2 + a_3$, when the user enters a_4, we only have to add this to the running sum $a_1 + a_2 + a_3 + a_4$

Running averages

```
double running_average() {
    double running_sum{0.0};

    for ( int count{1}; true; ++count ) {
        double value{};
        std::cout << "Enter a number: ";
        std::cin >> value;

        running_sum += value;

        std::cout << (running_sum/count) << std::endl;

        if ( value == 0.0 ) {
            return running_sum/count;
        }
    }
}
```

Expected output:

```
Enter a number: 5
5
Enter a number: 6
5.5
Enter a number: 7
6
Enter a number: 8
6.5
Enter a number: 9
7
Enter a number: 4
6.5
Enter a number: 3
6
Enter a number: 0
5.25
5.25
```


Running averages

- A more conventional solution:

```
double running_average() {
    double running_sum{0.0};
    int count{0};

    while ( true ) {
        double value{};
        std::cout << "Enter a number: ";
        std::cin >> value;

        running_sum += value;
        ++count;

        std::cout << (running_sum/count) << std::endl;

        if ( value == 0.0 ) {
            return running_sum/count;
        }
    }
}
```

Is even?

- A different type of function is a query:
  - Is an integer an even number?

```
bool is_even( int n );
```

Is even?

- A solution:

```
bool is_even( int n ) {
    if ( n%2 == 0 ) {
        return true;
    } else {
        return false;
    }
}
```

- An alternative solution:

```
bool is_even( int n ) {
    return ( n%2 == 0 );
}
```

Is even?

```
#include <iostream>

// Function declarations
int main();
bool is_even( int n );

// Function definitions
int main() {
    for ( int k{-5}; k <= 7; ++k ) {
        if ( is_even( k ) ) {
            std::cout << k << std::endl;
        } else {
            std::cout << "\t" << k << std::endl;
        }
    }

    return 0;
}

// ...other function definitions here
```

Expected output:

```
    -5
-4
    -3
-2
    -1
0
    1
2
    3
4
    5
6
    7
```

Is prime?

- Here is a different query:
  - Is a number a prime number?

```
bool is_prime( int n );
```

  - If $n < 2$, return `false`

Is prime?

- A solution:

```
bool is_prime( int n ) {
    if ( n == 2 ) {
        return true;
    } else if ( (n <= 1) || is_even( n ) ) {
        return false;
    } else {
        // If 'n' is divisible by any odd number >= 2,
        //    then 'n' is not prime
        // We only have to test divisibility so long as k*k <= n
        for ( int k{3}; k*k <= n; k += 2 ) {
            if ( n%k == 0 ) {
                return false;
            }
        }

        return true;
    }
}
```


Is prime?

```
#include <iostream>

// Function declarations
int main();
bool is_prime( int n );

// Function definitions
int main() {
    for ( int k{-30}; k <= 30; ++k ) {
        if ( is_prime( k ) ) {
            std::cout << k << std::endl;
        }
    }

    return 0;
}

// ...other function definitions
```

Expected output:

```
2
3
5
7
11
13
17
19
23
29
```


Reflections

- Author a function that prints a number, a pipe, and the mirror reflection of that number
  - For example

```
1970|0791
6857550|0557586
-42|24-
```

- This function does not return anything,
  so we indicate this with the return type `void`

```
void reflect( int n );
```

- If you want to return from a void function, just use

```
return;
```


Hint

- What is the output of the following?

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    int n{1967};
    std::cout << n%10 << std::endl;
    n /= 10;
    std::cout << n << std::endl;

    return 0;
}
```

Reflections

- A solution:

```
void reflect( int n ) {
    if ( n == 0 ) {
        std::cout << "0|0" << std::endl;
        return;
    }

    std::cout << n << "|";

    bool is_negative{false};

    if ( n < 0 ) {
        n = -n;
        is_negative = true;
    }
```


Reflections

```
    while ( n != 0 ) {
        std::cout << (n%10);
        n /= 10;
    }

    if ( is_negative ) {
        std::cout << "-";
    }

    std::cout << std::endl;

    return;
}
```


Reflections

- My test:

```
#include <iostream>

// Function declarations
int main();
void reflect( int n );

// Function definitions
int main() {
    for ( int k{-5135}; k < 10000; k += 733 ) {
        reflect( k );
    }

    return 0;
}

void reflect( int n ) {
    // Function body here...
}
```

Expected output:

```
-5135|5315-
-4402|2044-
-3669|9663-
-2936|6392-
-2203|3022-
-1470|0741-
-737|737-
-4|4-
729|927
1462|2641
2195|5912
2928|8292
3661|1663
4394|4934
5127|7215
5860|0685
6593|3956
7326|6237
8059|9508
8792|2978
9525|5259
```


Summary

- Following this lesson, you now:
  - Have implemented a number of functions
  - Have written tests for each of these functions
  - Have, if you did not do the above and did not immediately know the answers, prepared your Notice of Withdrawal to be submitted to the Registrar's Office
  - Should have much more confidence in your ability to author C++ code to solve specific problems

References

[1] Cplusplus.com
http://www.cplusplus.com/reference/cmath/

Acknowledgments

None so far.

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