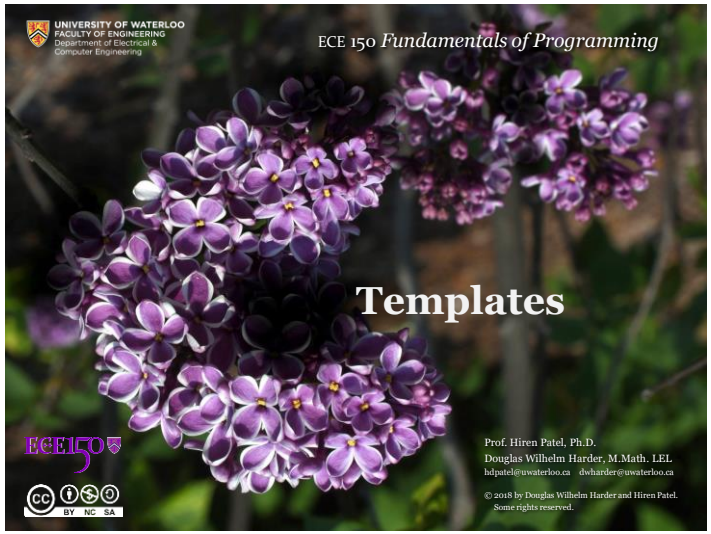

Outline

- In this lesson, we will:
  - Observe that much functionality does not depend on specific types
  - See that normally different types require different function definitions
  - Understand this is a serious waste of effort
  - Introduce templates and their use

Redundancy

- Suppose you author the function

```
std::size_t find_max( double const array[],
                      std::size_t const begin,
                      std::size_t const end ) {
    std::size_t index_max{begin};

    for ( std::size_t k{begin + 1}; k < end; ++k ) {
        if ( array[k] > array[index_max] ) {
            index_max = k;
        }
    }

    return index_max;
}
```


Redundancy

- Next, you must author another function:

```
std::size_t find_max( int const array[],
                      std::size_t const begin,
                      std::size_t const end ) {
    std::size_t index_max{begin};

    for ( std::size_t k{begin + 1}; k < end; ++k ) {
        if ( array[k] > array[index_max] ) {
            index_max = k;
        }
    }

    return index_max;
}
```

Redundancy

- And another:

```
std::size_t find_max( long const array[],
                      std::size_t const begin,
                      std::size_t const end ) {
    std::size_t index_max{begin};

    for ( std::size_t k{begin + 1}; k < end; ++k ) {
        if ( array[k] > array[index_max] ) {
            index_max = k;
        }
    }

    return index_max;
}
```


Redundancy

- And yet another:

```
std::size_t find_max( float const array[],
                      std::size_t const begin,
                      std::size_t const end ) {
    std::size_t index_max{begin};

    for ( std::size_t k{begin + 1}; k < end; ++k ) {
        if ( array[k] > array[index_max] ) {
            index_max = k;
        }
    }

    return index_max;
}
```


Redundancy

- Or you need a max function:

```
double max( double a, double b ) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}

long max( long a, long b ) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}

int max( int a, int b ) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}

float max( float a, float b ) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```


Redundancy

- The same goes with all the linear search algorithms:

```
std::size_t linear_search( double const array[],
                           std::size_t const begin,
                           std::size_t const end,
                           double const sought_value ) {
    for ( std::size_t k{begin}; k < end; ++k ) {
        if ( array[k] == sought_value ) {
            return k;
        }
    }

    return end;
}

std::size_t linear_search( int const array[],
                           std::size_t const begin,
                           std::size_t const end,
                           int const sought_value ) {
    for ( std::size_t k{begin}; k < end; ++k ) {
        if ( array[k] == sought_value ) {
            return k;
        }
    }

    return end;
}
```

Redundancy

- And the sorting algorithms:

```
void selection_sort( double array[],
                     std::size_t const begin,
                     std::size_t const end ) {
    for ( std::size_t k{end - 1}; k > begin; --k ) {
        std::size_t index_max{ find_max( array, begin, k + 1 ) };
        std::swap( array[index_max], array[k] );
    }
}
```

```
void selection_sort( int array[],
                     std::size_t const begin,
                     std::size_t const end ) {
    for ( std::size_t k{end - 1}; k > begin; --k ) {
        std::size_t index_max{ find_max( array, begin, k + 1 ) };
        std::swap( array[index_max], array[k] );
    }
}
```


Redundancy

- The only difference between all of these functions is the type
  - All other aspects are identical
- Could we not just write these functions once?
  - Templates to the rescue

Templates

- Consider the max function:
  - The only difference is the type

```
typename max( typename a, typename b );

typename max( typename a, typename b ) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```


Templates

- We can make the type a *parameter* of a template
  - A template is

```
template <typename T>
T max( T a, T b );

template <typename T>
T max( T a, T b ) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}
```

Definition (Google):
A shaped piece of metal, wood, card, plastic, or other material used as a pattern for processes such as painting, cutting out, shaping, or drilling.
– something that serves as a model for others to copy.

Using templated functions

- The compiler will now look at the type of the arguments:

```
#include <iostream>

// Function declarations
template <typename T>
T max( T a, T b );
int main();

// Function definitions
int main() {
    long a{5928395425359233}, b{5928395425359234};
    double x{3.14}, y{2.73};

    std::cout << max( a, b ) << std::endl;
    std::cout << max( x, y ) << std::endl;

    return 0;
}
```


Specifying the type

- The compiler, however, will not attempt to cast:

```
int main() {
    long a{5928395425359233};
    double x{3.14};

    std::cout << max( a, x ) << std::endl;

    return 0;
}
```

```
example.cpp: In function 'int main()':
example.cpp:21:28: error: no matching function for call to 'max(long int&, double&)'
     std::cout << max( a, y ) << std::endl;
                            ^
example.cpp:21:28: note: candidate is:
example.cpp:7:3: note: template<class T> T max(T, T)
 T max( T a, T b ) {
   ^
```


Specifying the type

- Instead, you can tell it which version to use:

```
int main() {
    long a{5928395425359233};
    double x{3.14};

    std::cout << max<double>( a, x ) << std::endl;

    return 0;
}
```

Output:

```
5.9284e+15
```


Swapping variables

- Consider this function:

```
void swap( int &a, int &b );

void swap( int &a, int &b ) {
    int tmp{a};
    a = b;
    b = tmp;
}
```

- Again, the functionality does not depend on the fact the arguments are of type `int`

Swapping variables

- Thus, we can *factor out* the type:

```
template <typename T>
void swap( T &a, T &b );

template <typename T>
void swap( T &a, T &b ) {
    T tmp{a};
    a = b;
    b = tmp;
}
```

Note that `tmp` is a local variable of the same type `T`

Finding the maximum array entry

- Again, here is another function we saw:

```
template <typename T>
std::size_t find_max( T const array[],
                      std::size_t const begin,
                      std::size_t const end ) {
    std::size_t index_max{begin};

    for ( std::size_t k{begin + 1}; k < end; ++k ) {
        if ( array[k] > array[index_max] ) {
            index_max = k;
        }
    }

    return index_max;
}
```


Selection sort

- Recall the selection sort function:

```
void selection_sort( double array[],
                     std::size_t const begin,
                     std::size_t const end ) {
    for ( std::size_t k{end - 1}; k > begin; --k ) {
        std::size_t index_max{ find_max( array, begin, k + 1 ) };
        std::swap( array[index_max], array[k] );
    }
}
```

- Note that the `std::swap` function doesn't care about it's arguments?

Selection sort

- Recall the selection sort function:

```
void selection_sort( double array[],
                     std::size_t const begin,
                     std::size_t const end ) {
    for ( std::size_t k{end - 1}; k > begin; --k ) {
        std::size_t index_max{ find_max( array, begin, k + 1 ) };
        std::swap( array[index_max], array[k] );
    }
}
```

- Note that the `std::swap` function doesn't care about it's arguments:
  - The standard library version uses templates:

```
template <typename T>
void swap( T &a, T &b ) {
    T c{a};
    a = b;
    b = c;
}
```

Selection sort

- Thus, we can make selection sort use templates:
  - Calls to `find_max` and `std::swap` also will use the correct types

```
template <typename T>
void selection_sort( T array[],
                     std::size_t const begin,
                     std::size_t const end ) {
    for ( std::size_t k{end - 1}; k > begin; --k ) {
        std::size_t index_max{ find_max( array, begin, k + 1 ) };
        std::swap( array[index_max], array[k] );
    }
}
```


Standard template library

- Many of these common functions already appear in the Standard Template Library (STL)

Function	Library
`std::swap`	`#include <utility>`
`std::max`	`#include <algorithm>`
`std::min`	`#include <algorithm>`

- Many of the algorithms we will cover in this course are implemented some way or another in the STL

Standard template library

- Other useful functions include `std::count`

```
#include <iostream>
#include <algorithm>

int main();

int main() {
    int array[10]{1, 5, 5, 2, 4, 7, 6, 2, 4, 5};

    for ( int k{0}; k < 10; ++k ) {
        std::cout << k << ": "
                  << std::count( array, array + 10, k )
                  << std::endl;
    }

    return 0;
}
```

Output:

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

Pointer arithmetic!
- The address of the first entry
- The address one beyond the last entry

Standard template library

- Other useful functions include `std::find`

```
#include <iostream>
#include <algorithm>
int main();

int main() {
    int array[10]{1, 5, 5, 2, 4, 7, 6, 2, 4, 5};

    for ( int k{0}; k < 10; ++k ) {
        int *p_loc = std::find( array, array + 10, k );

        if ( p_loc != array + 10 ) {
            std::cout << k << ": array[" << (p_loc - array) << "] == "
                      << *p_loc << std::endl;
        }
    }

    return 0;
}
```

Output:

```
1: array[0] == 1
2: array[3] == 2
4: array[4] == 4
5: array[1] == 5
6: array[6] == 6
7: array[5] == 7
```

Pointer difference!
- The number of positions between the two addresses

Standard template library

- Finding the minimum and maximum entries

```
#include <iostream>
#include <algorithm>

int main();

int main() {
    int array[10]{3, 5, 5, 2, 4, 7, 6, 2, 4, 5};

    int *p_min{ std::min_element( array, array + 10 ) };
    int *p_max{ std::max_element( array, array + 10 ) };

    std::cout << "Minimum: array[" << (p_min - array) << "] == "
              << *p_min << std::endl;
    std::cout << "Maximum: array[" << (p_max - array) << "] == "
              << *p_max << std::endl;

    return 0;
}
```

Output:

```
Minimum: array[3] == 2
Maximum: array[5] == 7
```

Pointer difference!

– The number of positions between the two addresses

Summary

- Following this lesson, you now
  - Understand the benefits of templates
  - Know that templates abstract out the type when the type is irrelevant to the operation
  - Know how to define a function to use a template
  - Understand that the Standard Template Library makes liberal use of templates for functions defined there

References

[1] No references?

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.