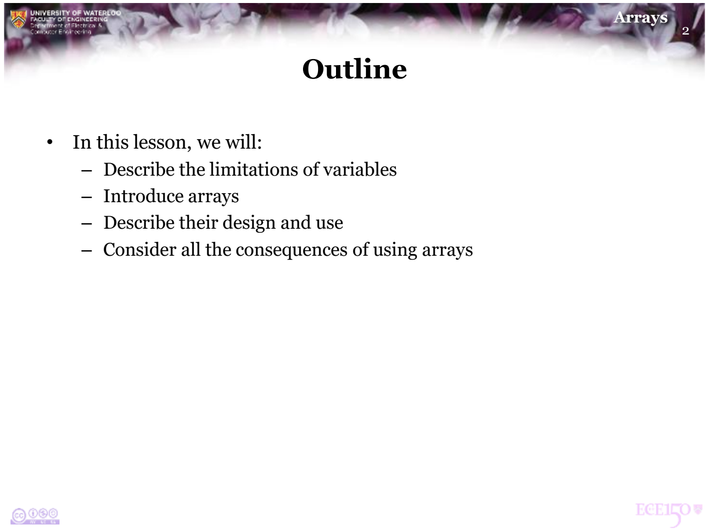
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

ECE 150 *Fundamentals of Programming*

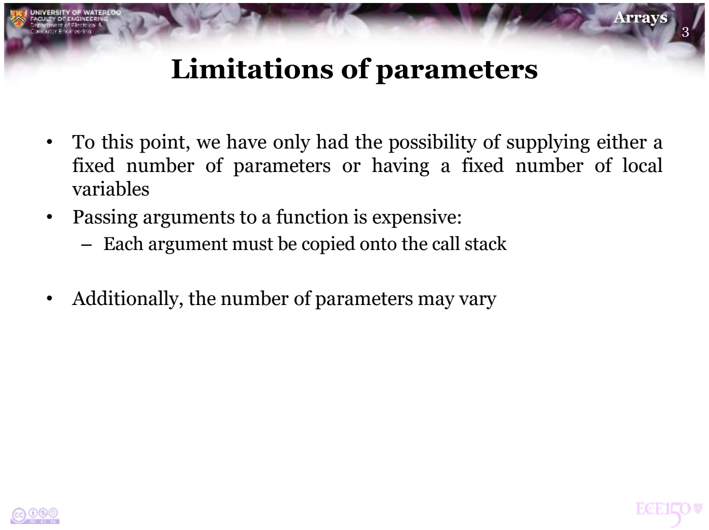
Arrays

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



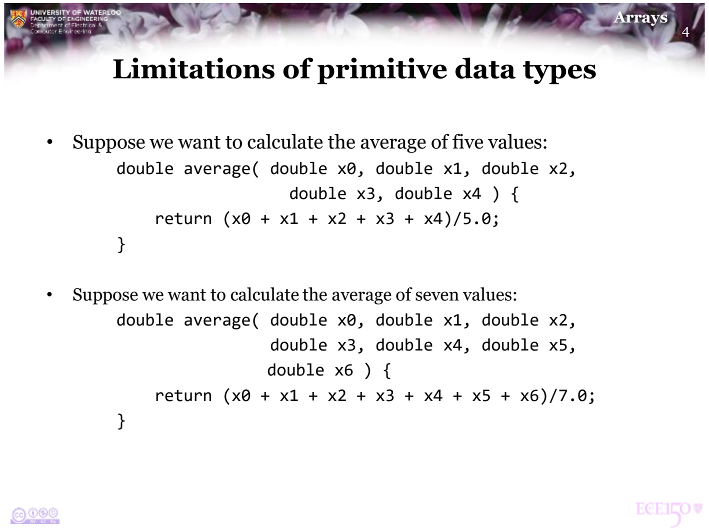
Outline

- In this lesson, we will:
  - Describe the limitations of variables
  - Introduce arrays
  - Describe their design and use
  - Consider all the consequences of using arrays



Limitations of parameters

- To this point, we have only had the possibility of supplying either a fixed number of parameters or having a fixed number of local variables
- Passing arguments to a function is expensive:
  - Each argument must be copied onto the call stack
- Additionally, the number of parameters may vary



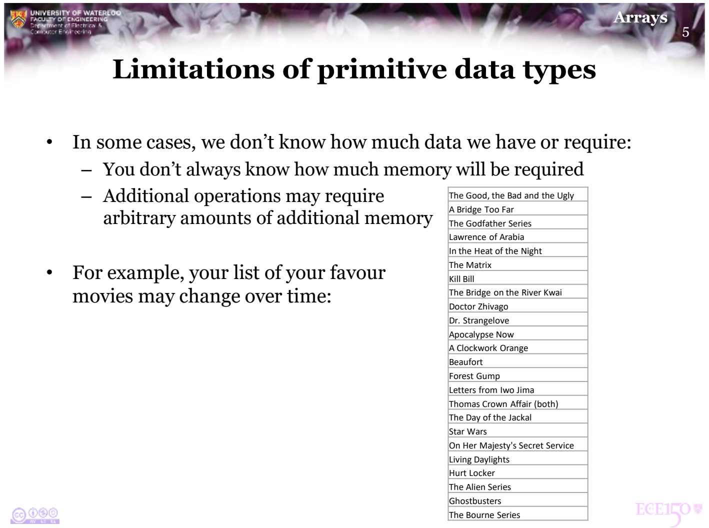
Limitations of primitive data types

- Suppose we want to calculate the average of five values:

```
double average( double x0, double x1, double x2,
                double x3, double x4 ) {
    return (x0 + x1 + x2 + x3 + x4)/5.0;
}
```

- Suppose we want to calculate the average of seven values:

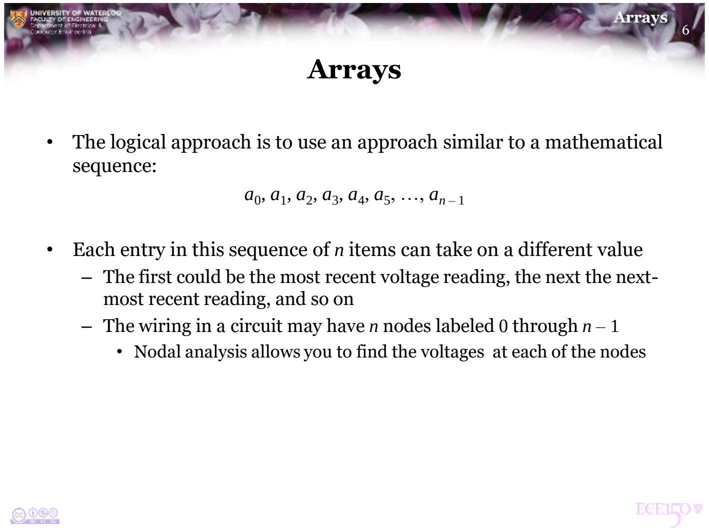
```
double average( double x0, double x1, double x2,
               double x3, double x4, double x5,
               double x6 ) {
    return (x0 + x1 + x2 + x3 + x4 + x5 + x6)/7.0;
}
```



Limitations of primitive data types

- In some cases, we don't know how much data we have or require:
  - You don't always know how much memory will be required
  - Additional operations may require arbitrary amounts of additional memory
- For example, your list of your favour movies may change over time:

The Good, the Bad and the Ugly
A Bridge Too Far
The Godfather Series
Lawrence of Arabia
In the Heat of the Night
The Matrix
Kill Bill
The Bridge on the River Kwai
Doctor Zhivago
Dr. Strangelove
Apocalypse Now
A Clockwork Orange
Beaufort
Forest Gump
Letters from Iwo Jima
Thomas Crown Affair (both)
The Day of the Jackal
Star Wars
On Her Majesty's Secret Service
Living Daylights
Hurt Locker
The Alien Series
Ghostbusters
The Bourne Series

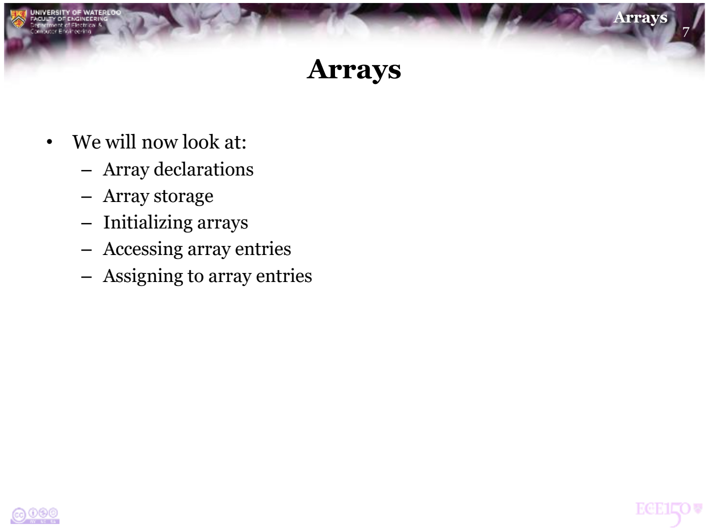


Arrays

- The logical approach is to use an approach similar to a mathematical sequence:

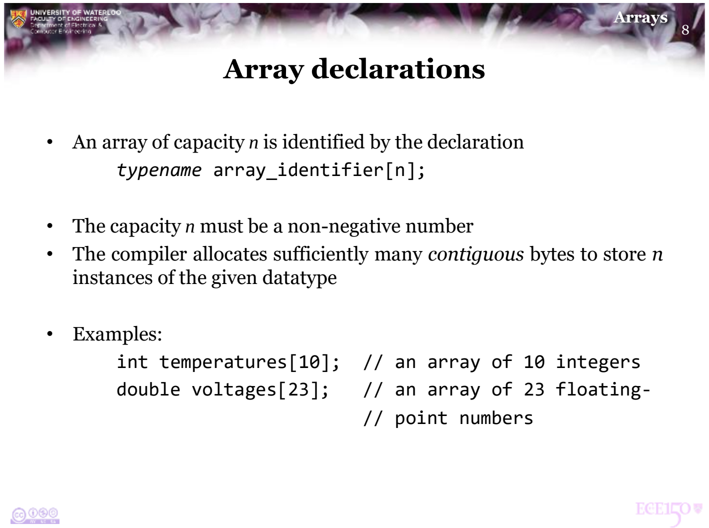
$$a_0, a_1, a_2, a_3, a_4, a_5, \ldots, a_{n-1}$$

- Each entry in this sequence of *n* items can take on a different value
  - The first could be the most recent voltage reading, the next the next-most recent reading, and so on
  - The wiring in a circuit may have *n* nodes labeled 0 through $n - 1$
    - Nodal analysis allows you to find the voltages at each of the nodes



Arrays

- We will now look at:
  - Array declarations
  - Array storage
  - Initializing arrays
  - Accessing array entries
  - Assigning to array entries



Array declarations

- An array of capacity *n* is identified by the declaration

```
typename array_identifier[n];
```

- The capacity *n* must be a non-negative number
- The compiler allocates sufficiently many *contiguous* bytes to store *n* instances of the given datatype
- Examples:

```
int temperatures[10];  // an array of 10 integers
double voltages[23];   // an array of 23 floating-
                       // point numbers
```

Array storage

- An array of 10 `int` requires 40 bytes
  - Each `int` requires 4 bytes

```
int temperatures[10];  // an array of 10 integers
```

- An array of 23 `double` requires 23 × 8 = 184 bytes
  - Each `double` requires 8 bytes

```
double voltages[23];   // an array of 23 floating-
                       // point numbers
```

Array entries

- The entries of an array store values of the given type and may be used like local variables
  - The entries of

    ```
    int data[4];  // an array of 4 integers
    ```

    are access with

    ```
    data[0]   data[1]   data[2]   data[3]
    ```

- The indices of

  ```
  datatype array_name[n];
  ```

  always go from `0` to `n - 1`

Array initialization

- Consider this uninitialized array:

```
int main() {
    double data[4];

    std::cout << data[0] << std::endl;
    std::cout << data[1] << std::endl;
    std::cout << data[2] << std::endl;
    std::cout << data[3] << std::endl;

    return 0;
}
```

These two, by chance, are zero

The output is

```
0
0
2.0733e-317
2.0731e-317
```

Array initialization

- This array has its four entries initialized:

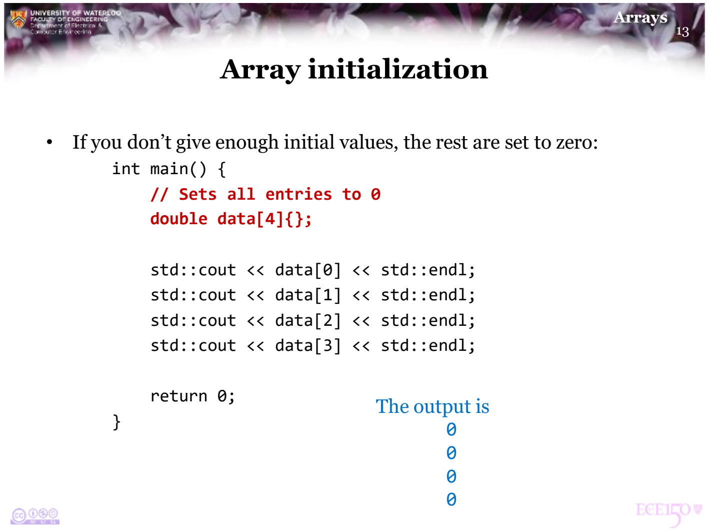
```
int main() {
    double data[4]{47.2, 48.3, 48.9, 49.4};

    std::cout << data[0] << std::endl;
    std::cout << data[1] << std::endl;
    std::cout << data[2] << std::endl;
    std::cout << data[3] << std::endl;

    return 0;
}
```

The output is

```
47.2
48.3
48.9
49.4
```



Array initialization

- If you don't give enough initial values, the rest are set to zero:

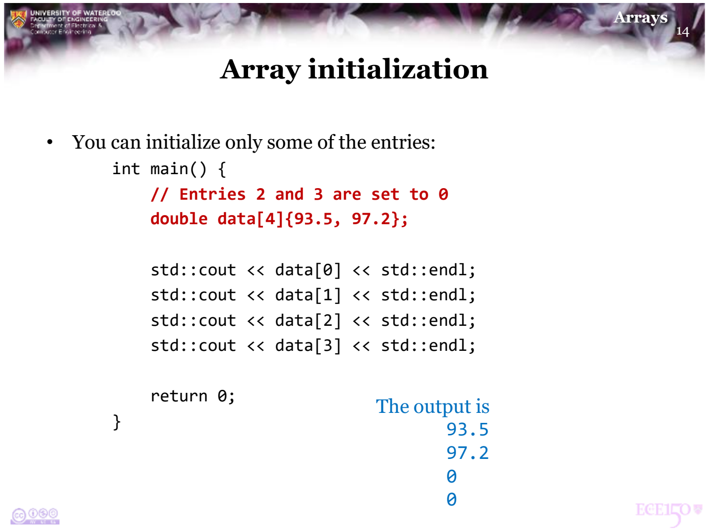
```
int main() {
    // Sets all entries to 0
    double data[4]{};

    std::cout << data[0] << std::endl;
    std::cout << data[1] << std::endl;
    std::cout << data[2] << std::endl;
    std::cout << data[3] << std::endl;

    return 0;
}
```

The output is

```
0
0
0
0
```



Array initialization

- You can initialize only some of the entries:

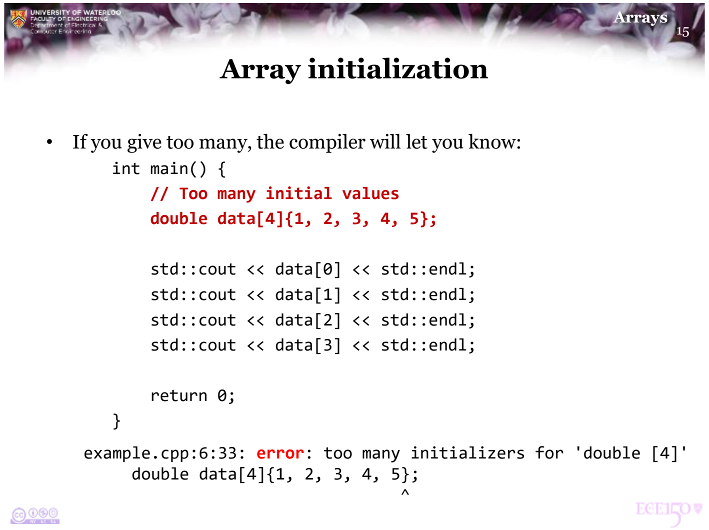
```
int main() {
    // Entries 2 and 3 are set to 0
    double data[4]{93.5, 97.2};

    std::cout << data[0] << std::endl;
    std::cout << data[1] << std::endl;
    std::cout << data[2] << std::endl;
    std::cout << data[3] << std::endl;

    return 0;
}
```

The output is

```
93.5
97.2
0
0
```



Array initialization

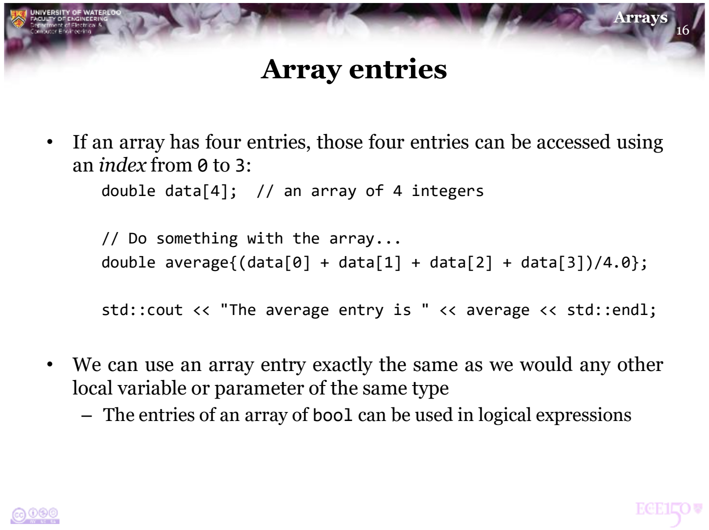
- If you give too many, the compiler will let you know:

```
int main() {
    // Too many initial values
    double data[4]{1, 2, 3, 4, 5};

    std::cout << data[0] << std::endl;
    std::cout << data[1] << std::endl;
    std::cout << data[2] << std::endl;
    std::cout << data[3] << std::endl;

    return 0;
}
```

```
example.cpp:6:33: error: too many initializers for 'double [4]'
     double data[4]{1, 2, 3, 4, 5};
                                 ^
```



Array entries

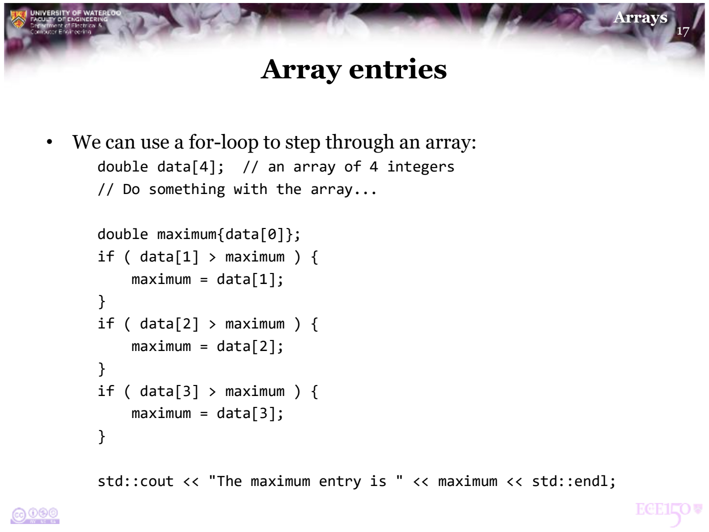
- If an array has four entries, those four entries can be accessed using an *index* from `0` to `3`:

```
double data[4];  // an array of 4 integers

// Do something with the array...
double average{(data[0] + data[1] + data[2] + data[3])/4.0};

std::cout << "The average entry is " << average << std::endl;
```

- We can use an array entry exactly the same as we would any other local variable or parameter of the same type
  - The entries of an array of `bool` can be used in logical expressions



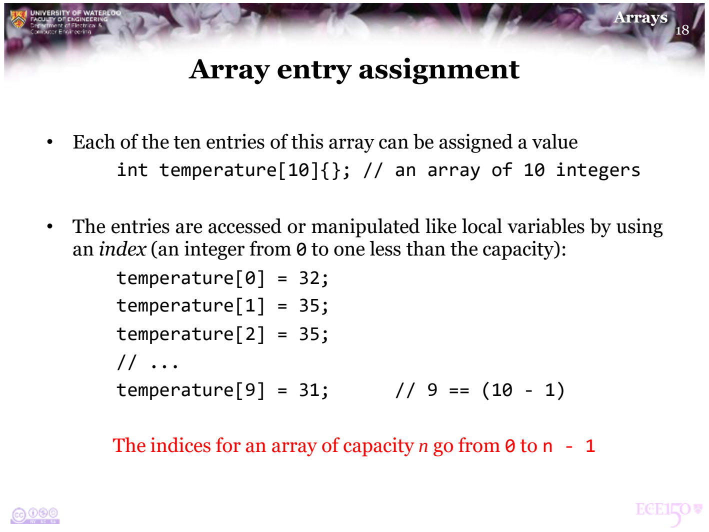
Array entries

- We can use a for-loop to step through an array:

```
double data[4];  // an array of 4 integers
// Do something with the array...

double maximum{data[0]};
if ( data[1] > maximum ) {
    maximum = data[1];
}
if ( data[2] > maximum ) {
    maximum = data[2];
}
if ( data[3] > maximum ) {
    maximum = data[3];
}

std::cout << "The maximum entry is " << maximum << std::endl;
```



Array entry assignment

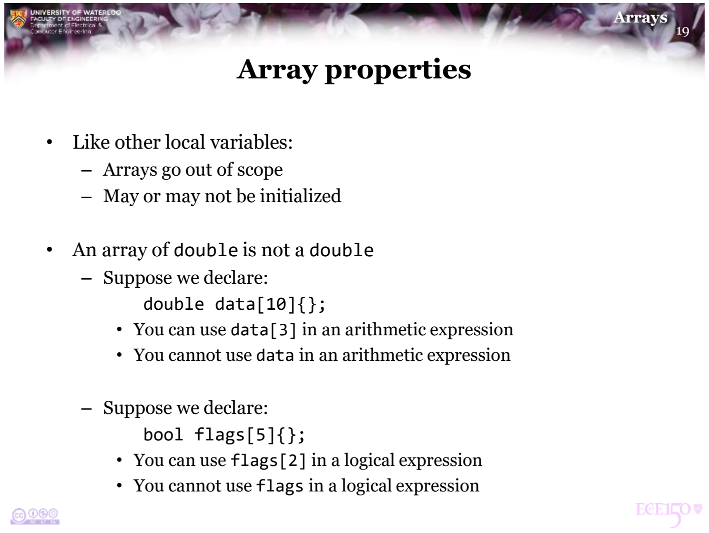
- Each of the ten entries of this array can be assigned a value

```
int temperature[10]{}; // an array of 10 integers
```

- The entries are accessed or manipulated like local variables by using an *index* (an integer from `0` to one less than the capacity):

```
temperature[0] = 32;
temperature[1] = 35;
temperature[2] = 35;
// ...
temperature[9] = 31;      // 9 == (10 - 1)
```

The indices for an array of capacity *n* go from `0` to `n - 1`



Array properties

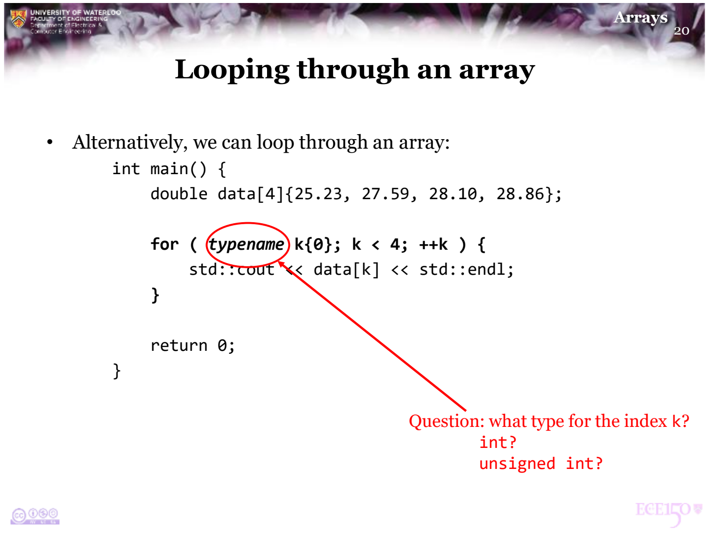
- Like other local variables:
  - Arrays go out of scope
  - May or may not be initialized
- An array of `double` is not a `double`
  - Suppose we declare:

    ```
    double data[10]{};
    ```

    - You can use `data[3]` in an arithmetic expression
    - You cannot use `data` in an arithmetic expression
  - Suppose we declare:

    ```
    bool flags[5]{};
    ```

    - You can use `flags[2]` in a logical expression
    - You cannot use `flags` in a logical expression



Looping through an array

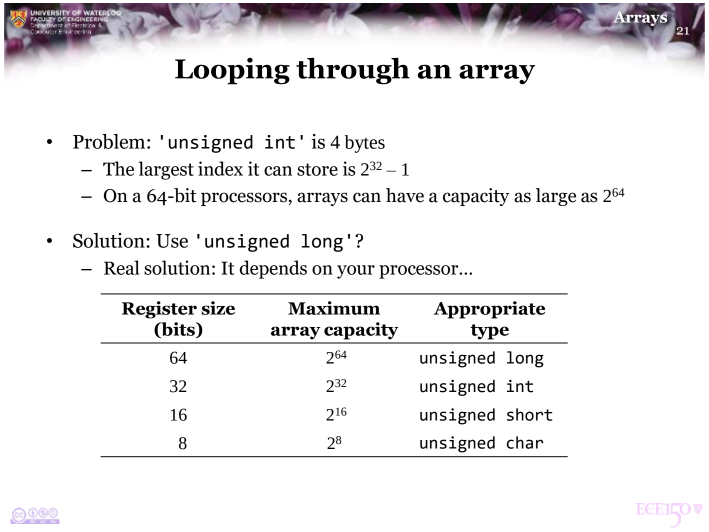
- Alternatively, we can loop through an array:

```
int main() {
    double data[4]{25.23, 27.59, 28.10, 28.86};

    for ( typename k{0}; k < 4; ++k ) {
        std::cout << data[k] << std::endl;
    }

    return 0;
}
```

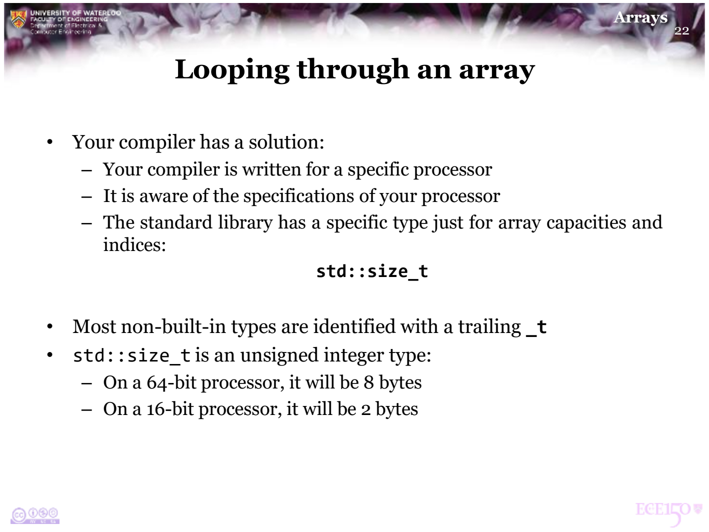
Question: what type for the index k?
`int?`
`unsigned int?`



Looping through an array

- Problem: `'unsigned int'` is 4 bytes
  - The largest index it can store is $2^{32} - 1$
  - On a 64-bit processors, arrays can have a capacity as large as 2^{64}
- Solution: Use `'unsigned long'`?
  - Real solution: It depends on your processor…

Register size (bits)	**Maximum array capacity**	**Appropriate type**
64	2^{64}	`unsigned long`
32	2^{32}	`unsigned int`
16	2^{16}	`unsigned short`
8	2^{8}	`unsigned char`

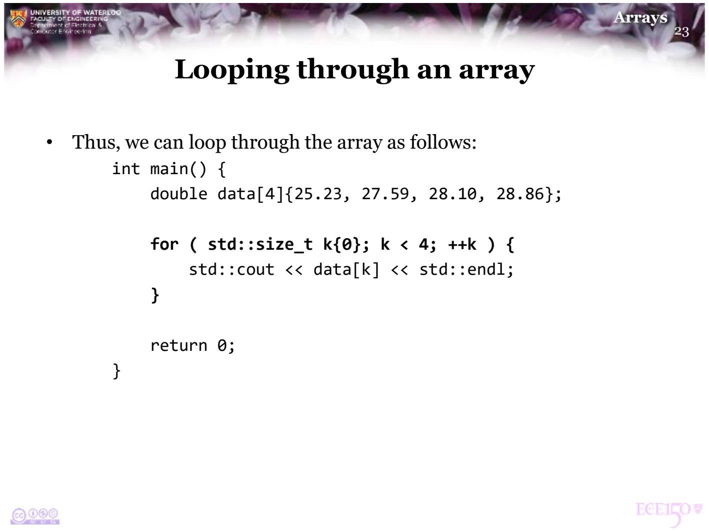


Looping through an array

- Your compiler has a solution:
  - Your compiler is written for a specific processor
  - It is aware of the specifications of your processor
  - The standard library has a specific type just for array capacities and indices:

`std::size_t`

- Most non-built-in types are identified with a trailing **`_t`**
- `std::size_t` is an unsigned integer type:
  - On a 64-bit processor, it will be 8 bytes
  - On a 16-bit processor, it will be 2 bytes



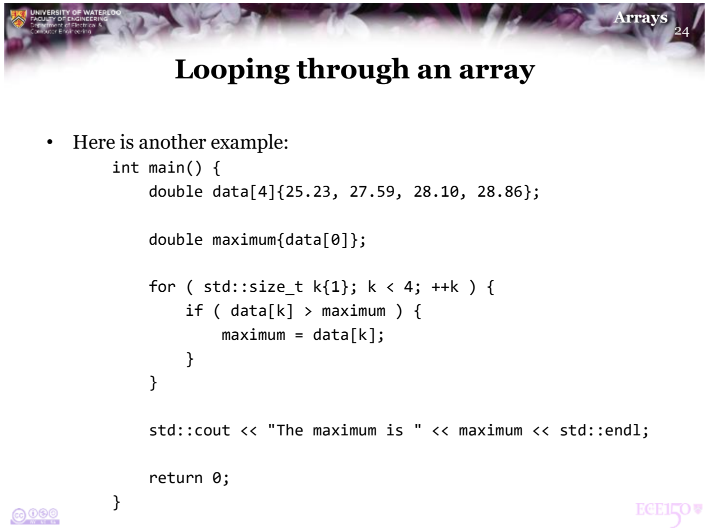
Looping through an array

- Thus, we can loop through the array as follows:

```
int main() {
    double data[4]{25.23, 27.59, 28.10, 28.86};

    for ( std::size_t k{0}; k < 4; ++k ) {
        std::cout << data[k] << std::endl;
    }

    return 0;
}
```



Looping through an array

- Here is another example:

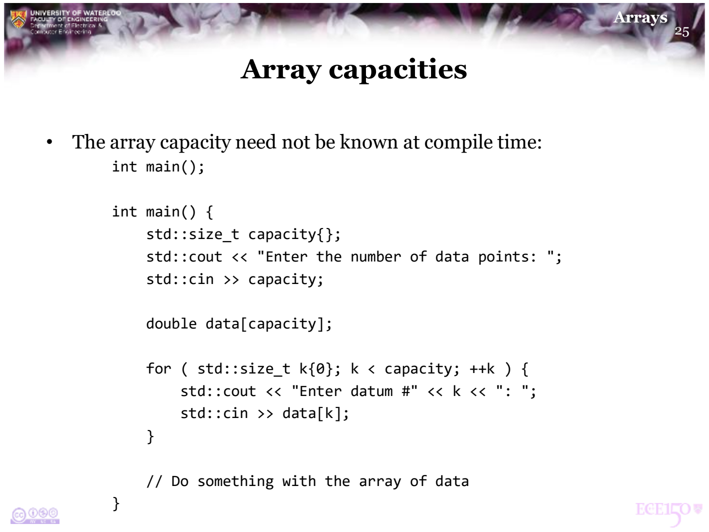
```
int main() {
    double data[4]{25.23, 27.59, 28.10, 28.86};

    double maximum{data[0]};

    for ( std::size_t k{1}; k < 4; ++k ) {
        if ( data[k] > maximum ) {
            maximum = data[k];
        }
    }

    std::cout << "The maximum is " << maximum << std::endl;

    return 0;
}
```



Array capacities

- The array capacity need not be known at compile time:

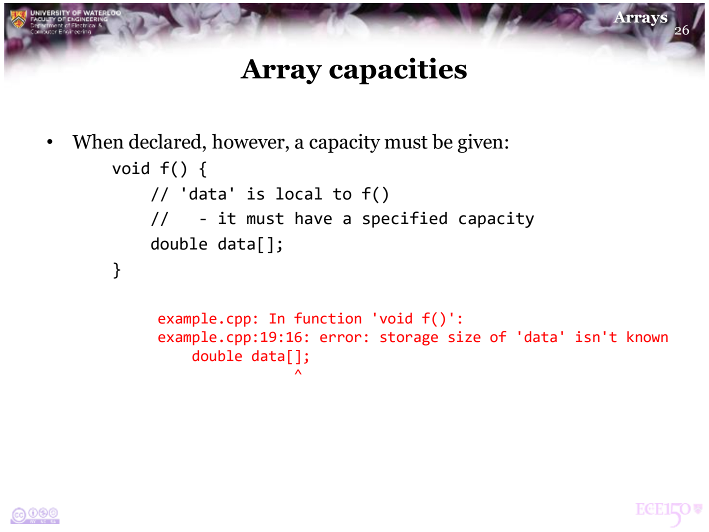
```
int main();

int main() {
    std::size_t capacity{};
    std::cout << "Enter the number of data points: ";
    std::cin >> capacity;

    double data[capacity];

    for ( std::size_t k{0}; k < capacity; ++k ) {
        std::cout << "Enter datum #" << k << ": ";
        std::cin >> data[k];
    }

    // Do something with the array of data
}
```

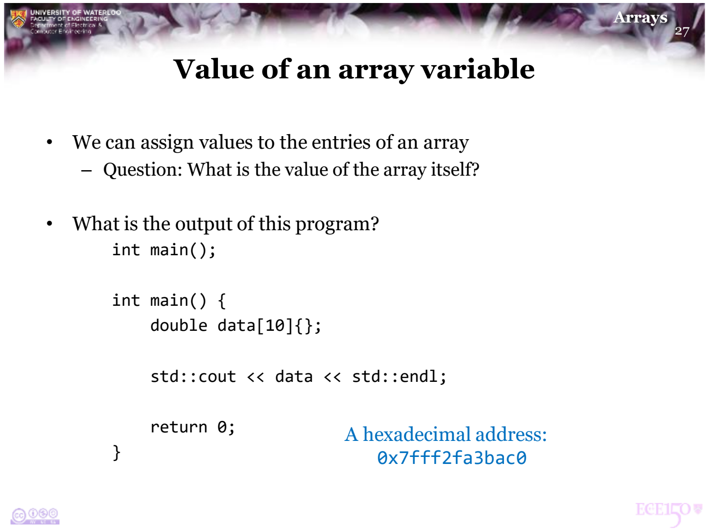


Array capacities

- When declared, however, a capacity must be given:

```
void f() {
    // 'data' is local to f()
    //   - it must have a specified capacity
    double data[];
}
```

```
example.cpp: In function 'void f()':
example.cpp:19:16: error: storage size of 'data' isn't known
    double data[];
                ^
```



Value of an array variable

- We can assign values to the entries of an array
  - Question: What is the value of the array itself?

- What is the output of this program?

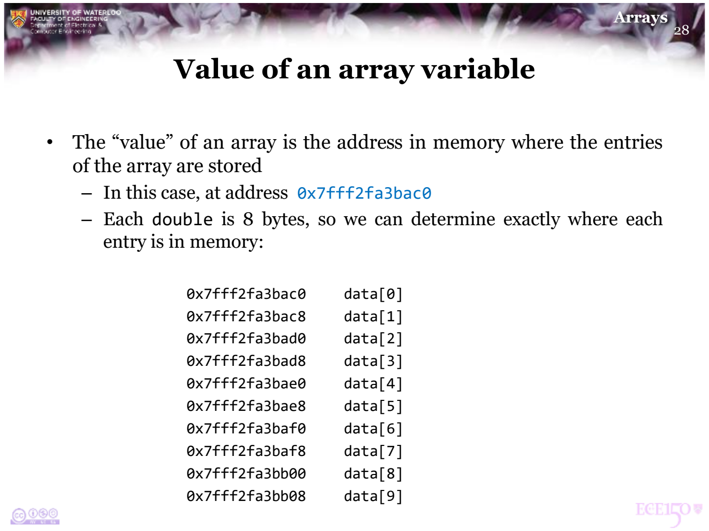
```
int main();

int main() {
    double data[10]{};

    std::cout << data << std::endl;

    return 0;
}
```

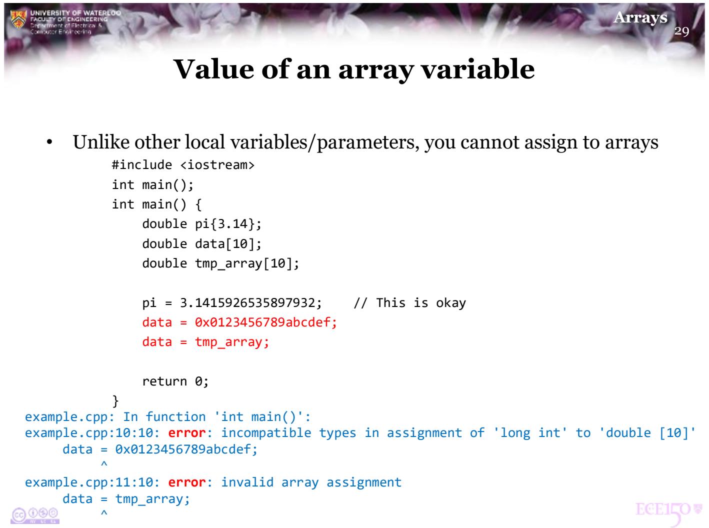
A hexadecimal address:
`0x7fff2fa3bac0`



Value of an array variable

- The "value" of an array is the address in memory where the entries of the array are stored
  - In this case, at address `0x7fff2fa3bac0`
  - Each `double` is 8 bytes, so we can determine exactly where each entry is in memory:

0x7fff2fa3bac0	data[0]
0x7fff2fa3bac8	data[1]
0x7fff2fa3bad0	data[2]
0x7fff2fa3bad8	data[3]
0x7fff2fa3bae0	data[4]
0x7fff2fa3bae8	data[5]
0x7fff2fa3baf0	data[6]
0x7fff2fa3baf8	data[7]
0x7fff2fa3bb00	data[8]
0x7fff2fa3bb08	data[9]



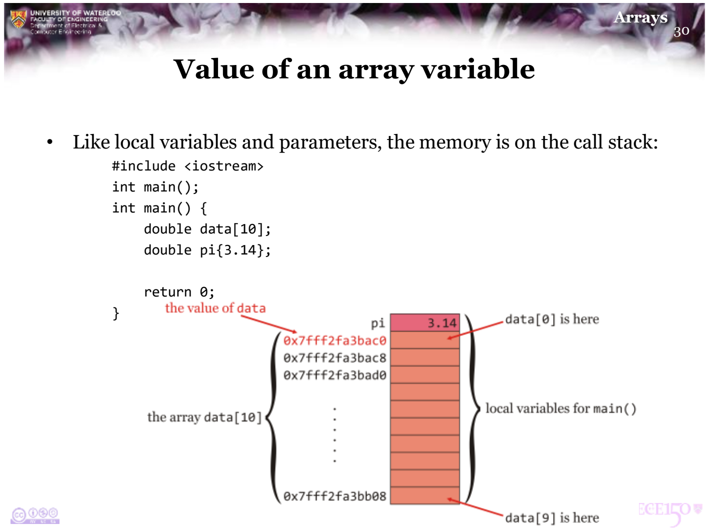
Value of an array variable

- Unlike other local variables/parameters, you cannot assign to arrays

```
        #include <iostream>
        int main();
        int main() {
            double pi{3.14};
            double data[10];
            double tmp_array[10];

            pi = 3.1415926535897932;    // This is okay
            data = 0x0123456789abcdef;
            data = tmp_array;

            return 0;
        }
example.cpp: In function 'int main()':
example.cpp:10:10: error: incompatible types in assignment of 'long int' to 'double [10]'
     data = 0x0123456789abcdef;
          ^
example.cpp:11:10: error: invalid array assignment
     data = tmp_array;
          ^
```



Value of an array variable

- Like local variables and parameters, the memory is on the call stack:

```
#include <iostream>
int main();
int main() {
    double data[10];
    double pi{3.14};

    return 0;
}
```

the value of data

pi

3.14

0x7fff2fa3bac0

0x7fff2fa3bac8

0x7fff2fa3bad0

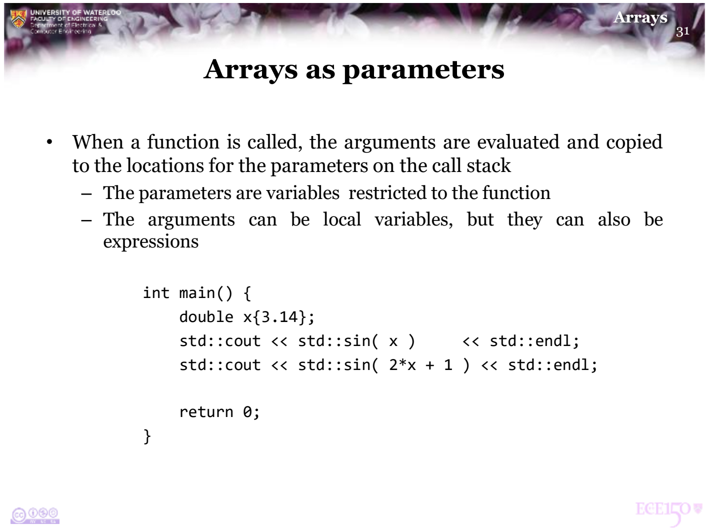
the array data[10]

0x7fff2fa3bb08

data[0] is here

local variables for main()

data[9] is here

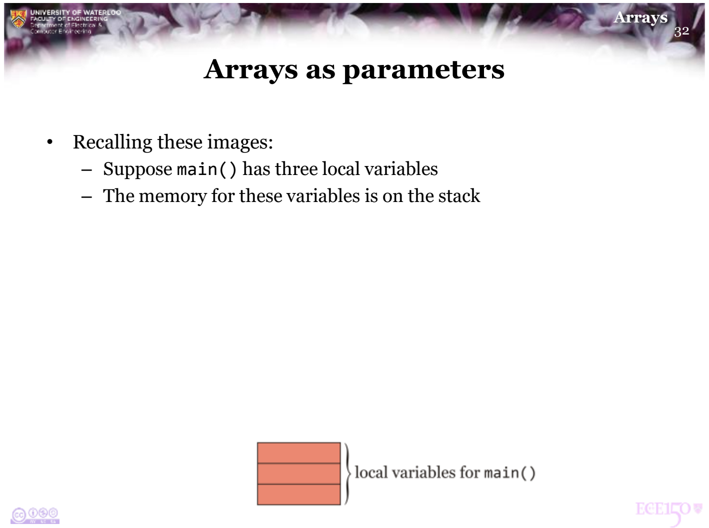


Arrays as parameters

- When a function is called, the arguments are evaluated and copied to the locations for the parameters on the call stack
  - The parameters are variables restricted to the function
  - The arguments can be local variables, but they can also be expressions

```
int main() {
    double x{3.14};
    std::cout << std::sin( x )       << std::endl;
    std::cout << std::sin( 2*x + 1 ) << std::endl;

    return 0;
}
```



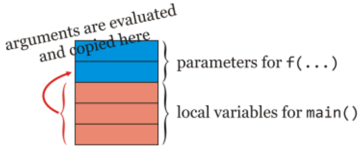
Arrays as parameters

- Recalling these images:
  - Suppose main() has three local variables
  - The memory for these variables is on the stack

local variables for main()

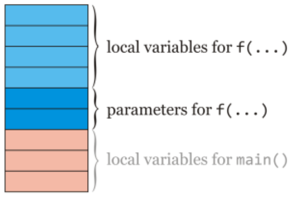
Arrays as parameters

- If `main()` calls `f(...)`, the arguments are evaluated and copied to the appropriate locations reserved for the parameters



Arrays as parameters

- When `f(...)` is called, additional space for any local variable for `f(...)` is also reserved on the stack
  - Inside `f(...)`, you can modify the parameters and local, but when the function exists, those changes are lost



Arrays as parameters

- We can write a function that accepts an array as a parameter

```
int main();
double average( double data[4] );

int main() {
    // drone speed in m/s
    double speeds[4]{178.2, 182.5, 187.1, 191.6};

    std::cout << average( speeds ) << std::endl;

    return 0;
}
```

```
double average( double data[4] ) {
    double sum{0.0};

    for ( std::size_t k{0}; k < 4; ++k ) {
        sum += data[k];
    }

    return sum/4.0;
}
```

Arrays as parameters

- But what is copied to the parameter?

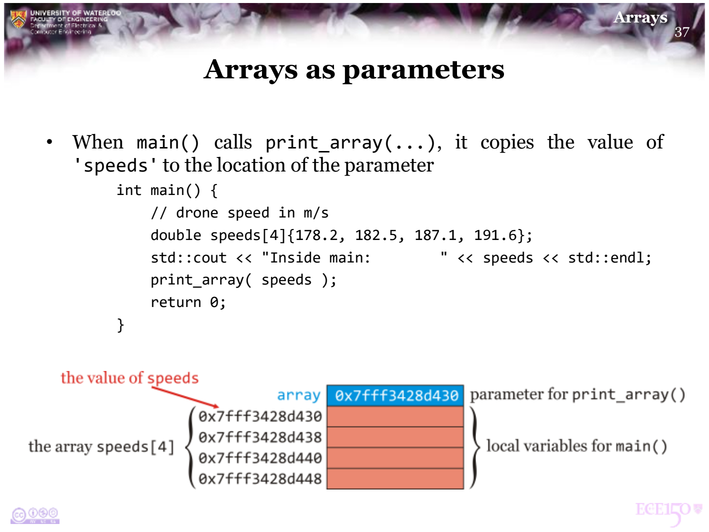
```
int main();
void print_array( double array[] );

int main() {
    // drone speed in m/s
    double speeds[4]{178.2, 182.5, 187.1, 191.6};
    std::cout << "Inside main:        " << speeds << std::endl;
    print_array( speeds );
    return 0;
}

void print_array( double array[] ) {
    std::cout << "Inside print_array: " << array << std::endl;
}
```

Output:

```
Inside main:        0x7fff3428d430
Inside print_array: 0x7fff3428d430
```



Arrays as parameters

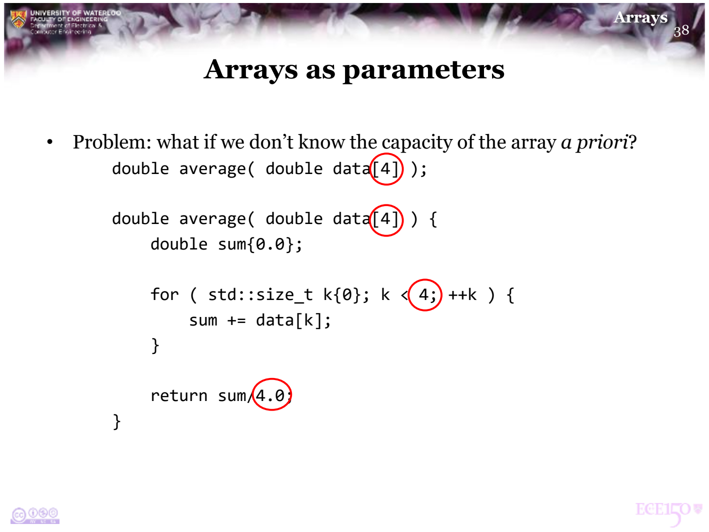
- When `main()` calls `print_array(...)`, it copies the value of `'speeds'` to the location of the parameter

```
int main() {
    // drone speed in m/s
    double speeds[4]{178.2, 182.5, 187.1, 191.6};
    std::cout << "Inside main:        " << speeds << std::endl;
    print_array( speeds );
    return 0;
}
```

the value of speeds

array 0x7fff3428d430 parameter for print_array()

the array speeds[4]
0x7fff3428d430
0x7fff3428d438
0x7fff3428d440
0x7fff3428d448
local variables for main()



Arrays as parameters

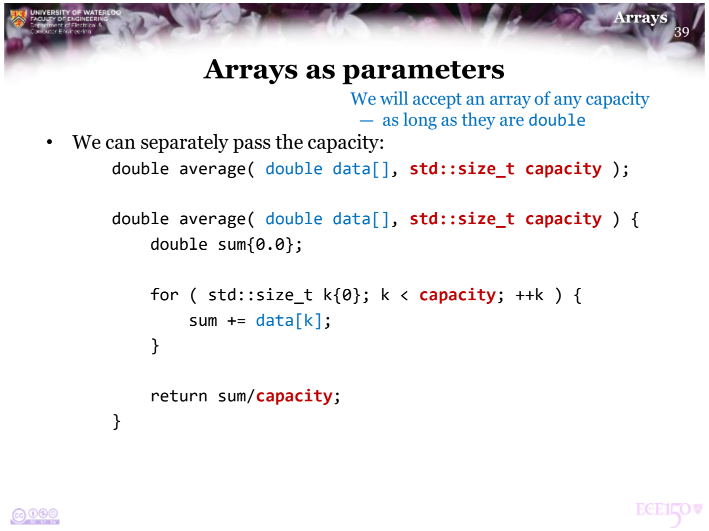
- Problem: what if we don’t know the capacity of the array *a priori*?

```
double average( double data[4] );

double average( double data[4] ) {
    double sum{0.0};

    for ( std::size_t k{0}; k < 4; ++k ) {
        sum += data[k];
    }

    return sum/4.0;
}
```



Arrays as parameters

We will accept an array of any capacity
— as long as they are double

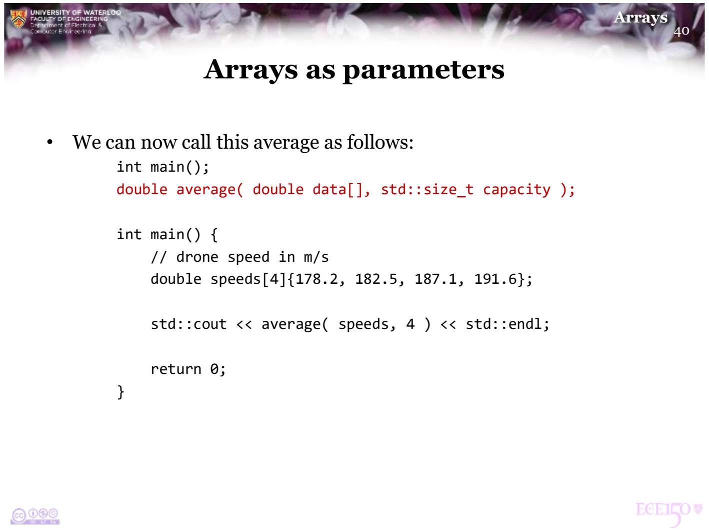
- We can separately pass the capacity:

```
double average( double data[], std::size_t capacity );

double average( double data[], std::size_t capacity ) {
    double sum{0.0};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        sum += data[k];
    }

    return sum/capacity;
}
```



Arrays as parameters

- We can now call this average as follows:

```
int main();
double average( double data[], std::size_t capacity );

int main() {
    // drone speed in m/s
    double speeds[4]{178.2, 182.5, 187.1, 191.6};

    std::cout << average( speeds, 4 ) << std::endl;

    return 0;
}
```

Arrays as parameters

- Suppose we author and then call this function:

```
double initialize( double array[], std::size_t capacity );

// Set all the entries of the array to 0.0
double initialize( double array[], std::size_t capacity ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        array[k] = 0.0;
    }
}
```

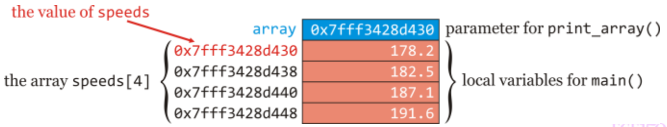
Arrays as parameters

- When we call this function

```
int main() {
    // drone speed in m/s
    double speeds[4]{178.2, 182.5, 187.1, 191.6};
    initialize_array( speeds, 4 );
    return 0;
}
```

the address of the array is copied to the parameter

  - When inside `initialize_array(...)`, we assign to `array[0]`, this changes the original array entry `speeds[0]`



Arrays as parameters

- Thus, the output of

```
// drone speed in m/s
double speeds[4]{178.2, 182.5, 187.1, 191.6};

for ( std::size_t k{0}; k < 4; ++k ) {
    std::cout << "speeds[" << k << "] = " speeds[k]
              << " m/s" << std::endl;
}

std::cout << std::endl;
initialize_array( speeds, 4 );

 for ( std::size_t k{0}; k < 4; ++k ) {
    std::cout << "speeds[" << k << "] = " speeds[k]
              << " m/s" << std::endl;
}
```

Output:

```
speeds[0] = 178.2 m/s
speeds[1] = 182.5 m/s
speeds[2] = 187.1 m/s
speeds[3] = 191.6 m/s

speeds[0] = 0 m/s
speeds[1] = 0 m/s
speeds[2] = 0 m/s
speeds[3] = 0 m/s
```

Exceeding array bounds

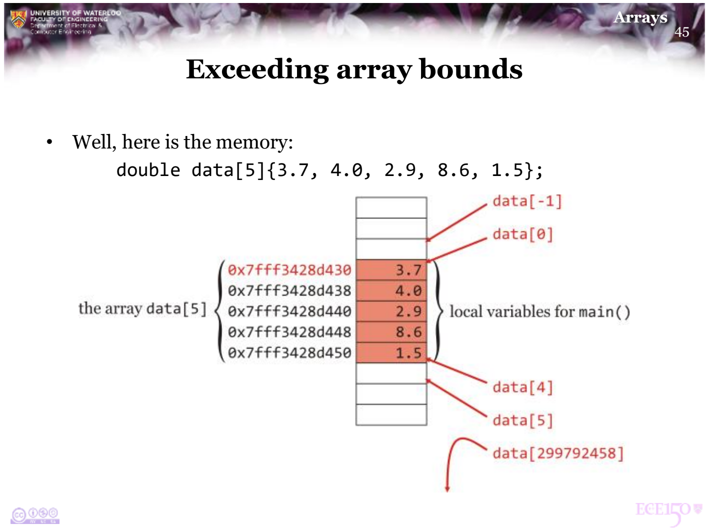
- The array

```
double data[5]{3.7, 4.0, 2.9, 8.6, 1.5};
```

has entries `data[0]` through `data[4]`

- Problem: What will happen if you try to access or assign to `data[-1]` or `data[5]` or even `data[299792458]`?
- Solution: It will just look in the appropriate location…

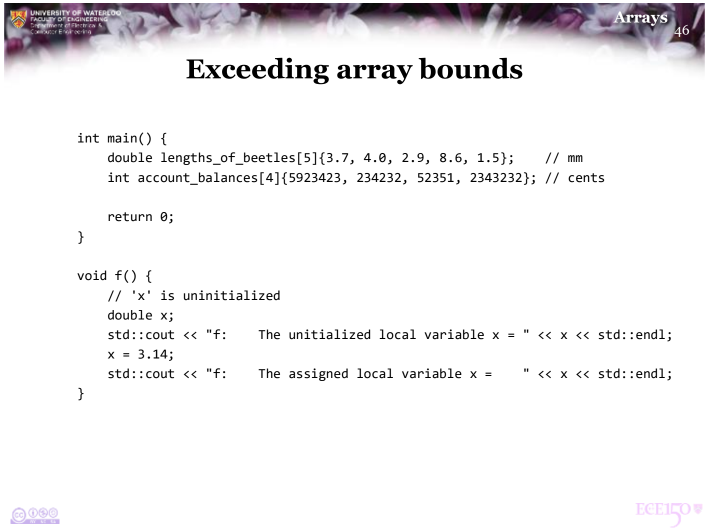
- Question: What is there?
- Answer: Other data including, but not limited to other local variables and other arrays



Exceeding array bounds

- Well, here is the memory:

```
double data[5]{3.7, 4.0, 2.9, 8.6, 1.5};
```

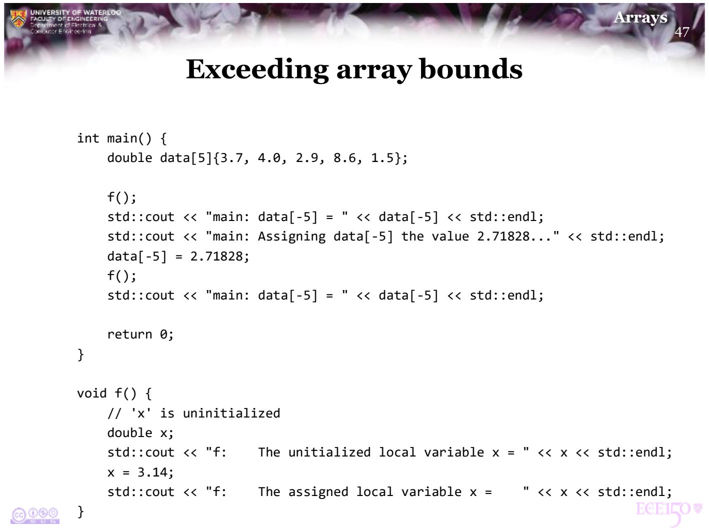


Exceeding array bounds

```
int main() {
    double lengths_of_beetles[5]{3.7, 4.0, 2.9, 8.6, 1.5};    // mm
    int account_balances[4]{5923423, 234232, 52351, 2343232}; // cents

    return 0;
}

void f() {
    // 'x' is uninitialized
    double x;
    std::cout << "f:    The unitialized local variable x = " << x << std::endl;
    x = 3.14;
    std::cout << "f:    The assigned local variable x =    " << x << std::endl;
}
```



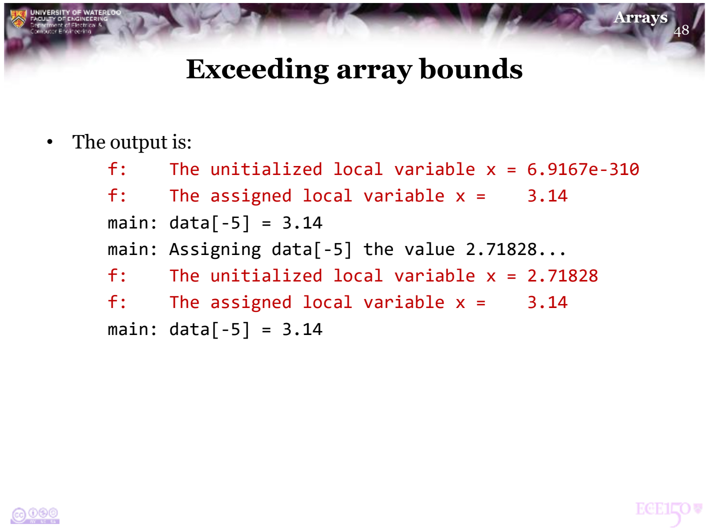
Exceeding array bounds

```
int main() {
    double data[5]{3.7, 4.0, 2.9, 8.6, 1.5};

    f();
    std::cout << "main: data[-5] = " << data[-5] << std::endl;
    std::cout << "main: Assigning data[-5] the value 2.71828..." << std::endl;
    data[-5] = 2.71828;
    f();
    std::cout << "main: data[-5] = " << data[-5] << std::endl;

    return 0;
}

void f() {
    // 'x' is uninitialized
    double x;
    std::cout << "f:    The unitialized local variable x = " << x << std::endl;
    x = 3.14;
    std::cout << "f:    The assigned local variable x =    " << x << std::endl;
}
```



Exceeding array bounds

- The output is:

```
f:    The unitialized local variable x = 6.9167e-310
f:    The assigned local variable x =    3.14
main: data[-5] = 3.14
main: Assigning data[-5] the value 2.71828...
f:    The unitialized local variable x = 2.71828
f:    The assigned local variable x =    3.14
main: data[-5] = 3.14
```

Exceeding array bounds

- How about this program?

```
#include <iostream>

int main();

int main() {
    double data[10]{3.7, 4.0, 2.9, 8.6, 1.5};

    std::cout << data[299792458] << std::endl;

    return 0;
}
```

Output:

```
  Segmentation fault (core dumped)
```

or some other catastrophic error…
  – The program execution is terminated

Exceeding array bounds

- The most common error:

```
void initialize( double array[], std::size_t capacity );

void initialize( double array[], std::size_t capacity ) {
    for ( std::size_t k{1}; k <= capacity; ++k ) {
        array[k] = 0.0;
    }
}
```

- Forgetting that an array of capacity 32 has entries indexed from 0 to 31 one of the most significant issues for novice programmers

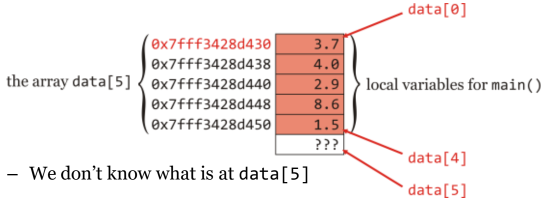
Exceeding array bounds

- Given this program:

```
int main() {
    double data[5]{3.7, 4.0, 2.9, 8.6, 1.5};

    initialize( data, 5 );
    return 0;
}
```

- The initialized memory for the array `data` is here



the array data[5]		local variables for main()
0x7fff3428d430	3.7	data[0]
0x7fff3428d438	4.0	
0x7fff3428d440	2.9	
0x7fff3428d448	8.6	
0x7fff3428d450	1.5	data[4]
	???	data[5]

  – We don't know what is at `data[5]`

Exceeding array bounds

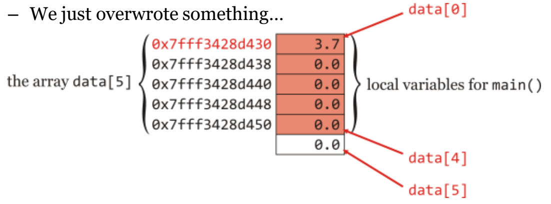
- Given this program:

```
int main() {
    double data[5]{3.7, 4.0, 2.9, 8.6, 1.5};

    initialize( data, 5 );
    return 0;
}
```

```
for ( std::size_t k{1}; k <= capacity; ++k ) {
    array[k] = 0.0;
}
```

- After we call `initialize(...)`, we have:
  – We just overwrote something…



the array data[5]		local variables for main()
0x7fff3428d430	3.7	data[0]
0x7fff3428d438	0.0	
0x7fff3428d440	0.0	
0x7fff3428d448	0.0	
0x7fff3428d450	0.0	data[4]
	0.0	data[5]

Summary

- Following this lesson, you now
  - Understand how to declare an array as a local variable and initialize its entries
  - Know how to access and assign to array entries
    - That array entries can be treated like local variables or parameters of the same type
    - Arrays cannot be used in arithmetic or logical expressions
  - Can step through an array with a for loop
  - Know that array variables are assigned the address in memory where that array is stored
  - Understand that arrays, if passed as arguments to a function, simply pass that address
    - Changing an array entry of a parameter changes the argument
  - Access entries outside the array bounds is dangerous

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.