Outline

- In this lesson, we will:
  - Describe the limitations of variables
  - Introduce local arrays
  - Look at initializing arrays
  - Describe their design and use
    - We will see a number of applications
  - Consider all the consequences of using arrays

Limitations of primitive data types

- To this point, we have only had the possibility of using:
  - A fixed number of parameters
  - A fixed number of local variables
- Each parameter or local variable must be separately declared

Limitations of primitive data types

- Suppose we want to query the user for three values:

```
int main() {
    int a1{};
    int a2{};
    int a3{};

    std::cout << "Enter an integer: ";
    std::cin >> a1;
    std::cout << "Enter a second integer: ";
    std::cin >> a2;
    std::cout << "Enter a third integer: ";
    std::cin >> a3;

    std::cout << "The average of these three is "
              << (a1 + a2 + a3)/3.0 << std::endl;

    return 0;
}
```

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

- 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
- 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
  - 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 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

```
std::cout << data[0] << data[1]
          << data[2] << data[3] << std::endl;
std::cout << (data[0] + data[1]
            + data[2] + data[3]) << std::endl;
```

- The indices of

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

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

Array initialization

- Consider this uninitialized array:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
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;
}
```

No {}

The output is

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

These two, by chance, are zero

Array initialization

- Instead, we can use a for loop and a loop variable to index the array:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double data[4];

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

    return 0;
}
```

The output is

```
3.1842e-314
2.12199e-314
2.12199e-314
0
```

This entry, by chance, is zero

Array initialization

- This array has its four entries initialized:

```
#include <iostream>

// Function declarations
int main();

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

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

    return 0;
}
```

The output is

```
47.2
48.3
48.9
49.4
```


Array initialization

- To initialize all entries to the default value, use {}:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double data[4]{};

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

    return 0;
}
```

The output is

```
0
0
0
0
```


Array initialization

- If there are insufficient initial values, the default value is used:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double data[4]{93.5, 97.2};

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

    return 0;
}
```

The output is

```
93.5
97.2
0
0
```

Array initialization

- Too many initial values results in a compile-time error

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double data[4]{93.5, 97.2, 96.3, 98.4, 97.9};

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

    return 0;
}
```

```
example.cpp:8:33: error: too many initializers for 'double [4]'
     double data[4]{93.5, 97.2, 96.3, 98.4, 97.9};
                                     ^
```


Initial capacity

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

```
// Function definitions
int main() {
    unsigned int n{};
    std::cout << "How many entries do you want? ";
    std::cin >> n;

    double data[n]{};   // All entries initialized to 0.0

    for ( int k{0}; k < n; ++k ) {
        std::cout << "Enter entry " << k << ": ";
        std::cin >> data[k];
    }

    return 0;
}
```


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

Applications

- For the next four applications,

  we will assume that we have an array with *n* entries:

```
// Function definitions
int main() {
    unsigned int n{};
    std::cout << "How many entries do you want? ";
    std::cin >> n;
    assert( n > 0 );

    double data[n]{};

    for ( int k{0}; k < n; ++k ) {
        std::cout << "Enter entry " << k << ": ";
        std::cin >> data[k];
    }

    // Carry on from here...
```

Applications

- Let us find the average value: $\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$

```
double sum{0.0};

for ( unsigned int k{0}; k < n; ++k ) {
    sum += data[k];
}

double average{ sum/n };

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


Applications

- Let us find the standard deviation value: $\sigma = \sqrt{\frac{1}{n}\sum_{k=1}^{n}\left(x_k - \bar{x}\right)^2}$

```
sum = 0.0;

for ( unsigned int k{0}; k < n; ++k ) {
    sum += (data[k] - average)*(data[k] - average);
}

double std_dev{ std::sqrt( sum/n ) };

std::cout << "The standard deviation is "
          << std_dev << std::endl;
```


Applications

- Let us find the minimum and maximum values:

```
double minimum{ data[0] };
double maximum{ data[0] };

for ( unsigned int k{1}; k < n; ++k ) {
    if ( data[k] < minimum ) {
        minimum = data[k];
    } else if ( data[k] > maximum ) {
        maximum = data[k];
    }
}

std::cout << "The range of the array is ["
          << minimum << ", " << maximum << "]" << std::endl;
```


Applications

- Let us find the maximum entry and swap it with the last:

```
    double maximum = data[0];
    unsigned int max_index{0};

    for ( unsigned int k{1}; k < n; ++k ) {
        if ( data[k] > maximum ) {
            maximum = data[k];
            max_index = k;
        }
    }

    // Swap the two entries
    double tmp{data[max_index]};
    data[max_index] = data[n - 1];
    data[n - 1] = tmp;
    return 0;
}
```

Implementation of arrays

- The array
  `double data[5]{3.7, 4.0, 2.9, 8.6, 1.5};`
  stores five double in contiguous memory

0	3.7
1	4.0
2	2.9
3	8.6
4	1.5

Exceeding array bounds

- Problem: What will happen if you try to access or assign to `data[-1]` or `data[5]` or even `data[299792458]`?
  - Other programming languages check to ensure you do not exceed the array bounds
  - C++ just goes to the corresponding location…

-2	?
-1	?
0	3.7
1	4.0
2	2.9
3	8.6
4	1.5
5	?
6	?

Exceeding array bounds

- One common mistake is to loop from 1 to n:

```
double sum{0.0};

for ( unsigned int k{1}; k <= n; ++k ) {
    sum += data[k];
}

double average{ sum/n };

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

-2	?
-1	?
0	3.7
1	4.0
2	2.9
3	8.6
4	1.5
5	?
6	?

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
  - Know you can step through an array with a for loop
  - Seen a number of applications with arrays
  - Understand accessing entries outside the array bounds is dangerous

References

[1] No references?

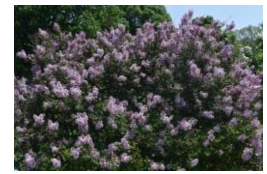
Colophon

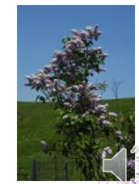
These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

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.