More on arrays

Arrays

- The previous topic covered arrays and their uses
  - Stores a fixed number of entries
  - Entries can be accessed with literal integers or integer data types
    - The index must be a value between 0 and capacity - 1
- This is sufficient for using arrays
- We will now investigate the implementation of arrays
  - This will lead to memory and addresses

Outline

- In this lesson, we will:
  - See how arrays are passed to functions
    - The behavior may be unexpected
  - Learn that array identifiers themselves have a value
  - See how that corresponds with passing of arrays to functions
  - See that local arrays are treated as const,
    but parameter arrays are not

Inability to assign arrays

- Unlike local variables of primitive data types,
  local arrays cannot be assigned to

```cpp
#include <iostream>

int main()
{
  int first_array[10]{};
  int second_array[10]{};

  first_array = second_array;

  return 0;
}
```

It is as if arrays identifiers are declared const

```cpp
example.cpp: In function ‘int main()’:
  example.cpp:11:19: error: invalid array assignment
    first_array = second_array;
```
const arrays

- However, if an array is declared const, its entries cannot be changed after initialization.

```cpp
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    int const primes[10]{2, 3, 5, 7, 11, 13, 17, 19, 23, 25};
    primes[9] = 29;
    return 0;
}
```

Inability to assign arrays

- It is a design feature
  - The assumption is that local arrays are temporary

- If you want to copy all the entries from one array to another, you must use a for loop:
  ```cpp
  for (int k = 0; k < capacity; ++k) {
    first_array[k] = second_array[k];
  }
  ```

Inability to return local arrays

- There is no mechanism to return a local array from a function

```cpp
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    int f()[5];
    int x[5]{3,4,5,6,7};
    return x;
}
```

Local arrays

- Local arrays are meant to temporarily store information for the duration of a function
  - C++ never creates new arrays and copies the entries over for you
  - We cannot return the local array, as it was local to the function
  - Later, we will see how we can:
    create, manipulate and return persistent arrays
Arrays

Passing arrays as arguments

• Local arrays can be passed as arguments to a function
  – You may expect notation a follows:
    // Function declarations;
    double average( double array[10] );
    int main();
  – Solution: the array capacity must be passed as a separate argument
    // Function declarations;
    double average( double array[], unsigned int capacity );
    int main();

The identifier of the array
The number of entries in the array

Passing arrays as arguments

• One solution is to have a separate local variable store the capacity:
  int main() {
    unsigned int const DATA_CAPACITY{5};
    double data[DATA_CAPACITY]{ 103.8, 105.1, 102.6, 103.7, 104.9 };
    std::cout << average( data, DATA_CAPACITY ) << std::endl;
    return 0;
  }

Passing arrays as arguments

• We calculate and print out the average:
  double average( double array[], unsigned int capacity ) {
    double sum{0.0};
    for ( unsigned int k{0}; k < capacity; ++k ) {
      sum += array[k];
    }
    return sum/capacity;
  }
  int main() {
    double data[5]{ 103.8, 105.1, 102.6, 103.7, 104.9 };
    std::cout << average( data, 5 ) << std::endl;
    return 0;
  }

Output: 104.02

Passing arrays as arguments

• This prints the entries of an array:
  void print_array( double array[], unsigned int capacity ) {
    if ( capacity == 0 ) {
      return;
    }
    std::cout << array[0];
    for ( unsigned int k{1}; k < capacity; ++k ) {
      std::cout << ", " << array[k]
    }
    std::cout << std::endl;
  }

Passing arrays as arguments

- This function converts an array into a string:

```
std::string to_string( double array[], unsigned int capacity ) {
    if ( capacity == 0 ) {
        return "";
    }
    std::string return_string = std::to_string( array[0] );
    for ( unsigned int k = 1; k < capacity; ++k ) {
        return_string += ", " + std::to_string( array[k] );
    }
    return return_string;
}
```

- Here is an interesting function:

```
void reset( double array[], unsigned int capacity ) {
    for ( unsigned int k = 0; k < capacity; ++k ) {
        array[k] = 0.0;
    }
}
```

```
int main() {
    unsigned int const DATA_CAPACITY(5);
    double data[DATA_CAPACITY] = { 103.8, 105.1, 102.6, 103.7, 104.9 };
    reset( data, DATA_CAPACITY );
    print_array( data, DATA_CAPACITY );
    return 0;
}
```

Output:
```
0, 0, 0, 0, 0
```

- Aren’t parameters passed by value?
  – Shouldn’t a copy of the array be sent?

- Problems:
  – Imagine copying an array with 1000 entries...

- Observation:
  – Like references, when passed, arrays cannot be part of an arithmetic or logical expression
  – You can only pass the array identifier
  • Consequently, differing behaviors should be expected
Passing arrays as arguments

• Question: what is passed?

```cpp
#include <iostream>

void some_function( double array[], unsigned int capacity ) {
    std::cout << array << std::endl;
}

int main() {
    double data[5]{ 103.8, 105.1, 102.6, 103.7, 104.9};
    std::cout << data << std::endl;
    some_function( data, 5 );
    return 0;
}
```

Output:

```
0xffffcb80
```

• Note that:
  – These seem to be hexadecimal numbers
  – The “values” of the local array and the parameter array are equal:
    `0xffffcb80`

• You may get different hexadecimal numbers,
  but both will be the same number

Array values

• Recall that an array of capacity \( n \) stores entries of the given type
  – Each entry occupies a fixed number of bytes

• Thus, the following arrays occupy:
  ```cpp
  int array_1[10]; // 10 x 4 = 40 bytes  
  double array_2[7]; // 7 x 8 = 56 bytes  
  bool array_3[100]; // 100 x 1 = 100 bytes
  ```

• The value of the array variable is the address of the first byte
  – It is the address in main memory

• Consider the following:
  ```cpp
  #include <iostream>
  int main() {
    int array_1[12]; // 48 bytes = 0x30 bytes  
    int array_2[20]; // 80 bytes = 0x50 bytes  
    int array_3[32]; // 208 bytes = 0x80 bytes
    std::cout << array_1 << std::endl;
    std::cout << array_2 << std::endl;
    std::cout << array_3 << std::endl;
    return 0;
  }
  ```

  The output is

```
0xffffcb80
```

+ 0x0

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80

+ 0xffffcb80
Arrays

Array values

- Looking at this graphically:
  - The output is:
    - 0xffffcbf0
    - 0xffffcbe0
    - 0xffffcb80
    - 0xffffcab0
    - 0xffffcab0
    - 0xffffcb80
    - 0xffffcbd0

208 bytes

80 bytes

48 bytes

Array addresses are passed by value

- When an array is passed as an argument, the value of the address is passed:

```cpp
void f(int param[], unsigned int capacity) {
    int local[5] = {1, 10, 100, 1000, 10000};
    std::cout << "local = " << local << std::endl;
    std::cout << "param = " << param << std::endl;
    Assigning param = local;
    param[3] = 1000;
    data = 0xffffcbf0;
}
```

Output:

- data = 0xffffcbf0
- param = 0xffffcbf0
- local = 0xffffcba0
- Assigning param = local;
- param[3] = 7
- param[3] = 1000
- data = 0xffffcbf0

- Parameters passed by value:
  - Parameters are passed by their addresses, not by their contents.

- Parameters can be modified:
  - Changes to the parameter array affect the corresponding entries in the argument array.

- This is different from passing arrays by value in other languages.

Summary

- Following this lesson, you now:
  - Understand that arrays store an address in memory
  - It is at that address that the array is stored
  - Understand that arrays, when passed to functions, are passed by this address value
  - A change to an entry in a parameter array changes the corresponding entry in the argument array
  - While local array identifiers cannot be assigned to, parameter array identifiers can be

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.