Arrays

Michael R. Nowak
University of Illinois Urbana-Champaign
Lesson Outline

1. Arrays

2. Limitations of arrays
Arrays

Why bother with arrays when we have std::vector?

- To understand how std::vector works, one must understand the compound type, array

C++ arrays have certain limitations and some strange behaviors that everyone programming in C++ should understand
Array declaration

We can declare an array of integers as follows:

```cpp
constexpr int x_size = 7;
int x[x_size];
```

The variable `x` is an array with 7 int entries

We can initialize an array using list initialization

```cpp
int y[] = {10, 2, 23, 14};
int y_size = 4;
```
The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time.

Array declaration

Array elements are accessed using subscripting

Given,

```cpp
constexpr int x_size = 7;
int x[x_size];
```

We access the \( i \)th element of \( x \) as \( x[i] \)

- The first element of \( x \) resides at \( x[0] \)
- The last element of \( x \) resides at \( x[6] \)
Arrays

The C++ standard specifies that the size of arrays must be constant and known at compile-time

```cpp
constexpr int x_size = 7;
int x[x_size];
```

Some compilers support runtime sizes,

```cpp
int size_x;
std::cin >> size_x;
int x[size_x];
```
Arrays

The C++ standard specifies that the size of arrays must be constant and known at compile-time

```cpp
constexpr int x_size = 7;
int x[x_size];
```

Some compilers support runtime sizes,

```cpp
int size_x;
std::cin >> size_x;
int x[size_x];
```

Avoid the temptation of specifying array sizes at runtime since not all compilers support it.
Arrays

Arrays do not know their own size

```cpp
constexpr int x_size = 7;
int x[x_size];
```

Therefore, writing `x.size()` will generate the compile-time error

```
error: member reference base type 'int [7]' is not a structure or union x.size();
```
Arrays

Arrays are a compound type built from the base type

- Similar to an int, they do not have any behaviors that you can ask of them

Since arrays do not know their own size, we always declare a variable that stores this information

```cpp
constexpr int x_size = 7;
int x[x_size];
```

and pass it around with the array
Arrays

-OR-

Since arrays do not know their own size, we use `#define`

```c
#define X_SIZE 7
```

```c
int x[X_SIZE];
```

and pass it around with the array.

How does this differ? Why do we care?
int main() {
    constexpr int size_x = 4;
    int x[size_x] = {1, 1, 2, 2};
    int sum = sum_elements(x, size_x);
    std::cout << sum << std::endl;
}

int sum_elements(int array[], int size) {
    int sum = 0;
    for (int i = 0; i < size; ++i)
        sum += array[i];
    return sum;
}
Arrays

Since the size of an array is known during compilation, we can compute it by dividing the byte size of the array by that of a single entry

```cpp
int x[] = {1, 1, 2, 2};
constexpr int size_x = sizeof(x) / sizeof(x[0]);
std::cout << size_x << std::endl;
```

However, this computation only works within the block in which the array is declared
Arrays

```cpp
int main() {
    int x[] = {1, 1, 2, 2};
    int sum = sum_elements(x);
}

int sum_elements(int array[]) {
    constexpr int size = sizeof(array) / sizeof(array[0]);
    std::cout << size << std::endl; // outputs 2
    int sum = 0;
    // execute sum procedure
    return sum;
}
```
Arrays

```cpp
int sum_elements(int array[]) {
    constexpr int size = sizeof(array) / sizeof(array[0]);
    std::cout << size << std::endl; // outputs 2

    int sum = 0;
    // execute sum procedure
    return sum;
}

int main() {
    int x[] = {1, 1, 2, 2};
    int sum = sum_elements(x);
}
```

In the cs128 environment, our compiler warns us of this:

- **warning**: `sizeof` on array function parameter will return size of 'int *' instead of 'int []'
  [-Wsizeof-array-argument]

- **warning**: `'sizeof (array)' will return the size of the pointer, not the array itself`
  [-Wsizeof-pointer-div]

Though a warning might not prohibit compilation...

```
clang++ -std=c++20 -Werror
```

would make these warnings an error (thereby prohibiting compilation)
Arrays

**warning**: `sizeof` on array function parameter will return size of 'int *' instead of 'int []' [-Wsizeof-array-argument]

**warning**: `sizeof (array)` will return the size of the pointer, not the array itself [-Wsizeof-pointer-div]

When passed to a function, an array decays into a pointer

- Even though the parameter is specified as `int array[]` it decays into `int* array`, losing any dimension information in the process
Arrays

Even though the parameter is specified as `int array[]` decays into `int* array`, we interact with an `int* array` the same way as an `int array[]`

▶ We access $i$th element of `array` as `array[i]`

▶ The first element of `x` resides at `x[0]`

▶ The seventh element of `x` at `x[6]`
This works because

\[ \text{array}[i] \]

is equivalent to pointer arithmetic

\[ \text{array}[i] = * (\text{array} + i) \]

Pointer arithmetic causes \( * (\text{array} + i) \) to be evaluated as

\[ \text{array} + \text{sizeof} (\text{int}) \times i \]

- We are simply adding an offset on to the base address to calculate a new address; we interpret the object at that address as an integer
Arrays

Something else to consider... observe the difference between these two blocks of code:

```cpp
int main() {
    int x[] = {1, 1, 2, 2};
    std::vector<int> y = {1, 1, 2, 2};
    foo_a(x);
    foo_v(y);

    std::cout << x[0] << 't' << y.at(0) << std::endl;
}

void foo_v(std::vector<int> f) {
    f.at(0) = 7;
}

void foo_a(int f[]) {
    f[0] = 7;
}
```
Lesson Outline

1. Arrays

2. Limitations of arrays
Limitations of arrays

The size of an array must be known at compile time

- How would we read in an unspecified number of values from a file? Right now,
  - Create a large array and potentially waste space
  - Create a small array and potentially not have enough space
  - Use an std::vector and leverage its push_back() behavior
- We will see how dynamic memory allocation allows us to create "resizable" arrays at runtime, exactly what std::vector does behind the scenes
Limitations of arrays

Arrays do not know their own size, so we're stuck passing around an extra variable with them

- We will see how classes allow us to group data together with the operations that act upon them

  - Eventually, we will see how `std::vector` is implemented to create the "illusion" of an ever growing collection of objects, storing attributes such as size and capacity, while providing an interface (behaviors) for us to interact with it
Limitations of arrays

Unlike std::vector's `at()` behavior, indices are not checked before accessing an array; arrays do not prevent us from accessing data outside their bounds

▶ In the best case, the program crashes immediately with a segmentation fault/violation communicating this

▶ Often, nothing will suggest an error and the program will keep running

▶ This can disrupt the integrity of your program's data and can lead to the computation of wrong results