FIRST LOOK AT ARRAYS

ADAM SWEENEY
CS 211

INTRODUCTION

- Consider pieces of information that are closely related
- They can be grouped under a single name in C++
  - Start with older C-style arrays

AGENDA

- What is Even an array?
- Declaring C-arrays (& initializing)
- Utilizing an array
- `const`

WHAT IS EVEN AN ARRAY

WHAT IS AN ARRAY?

- A mechanism to hold and access many variables of the same type through a single name
- Very useful for storing a collection of data
  - Makes operating on a set of data much simpler

ARRAYS IN MEMORY

- We should know by now how much memory simple data types require
- It makes sense that if we're storing 'n' variables of type 't', the space occupied is `n x (size of t in bytes)`
- What makes arrays special is that the memory is always contiguous

Type	Size (bytes)
char, bool	1
int	4
double	8

DECLARING ARRAYS (& INITIALIZING)

DECLARING C-STYLE ARRAYS

- It’s still a variable
- You still need a type, and a name
- Now, there’s a third part that makes it an array
- Here is an example declaration
  - `double grades[49];`
- That declares an array of doubles called grades, and it can hold 49 values

INITIALIZING ARRAYS

- It follows that the syntax for initializing arrays has a little extra to it as well
- Like any other variable, we can NOT know what our array contains unless we initialize it
- `char charArr[] = {‘a’, ‘b’, ‘c’};`
  - The size is optional if you are initializing; in this case the compiler sees three elementst, and sets the size to 3
- All or some elements can be initialized
- It can be tedious

NOTES

- There always needs to be a size
  - Explicitly between the []
  - Implicitly through initialization
- The size can NOT be a user-input value
  - The size must be known at compile-time
- The size can NOT be changed once declared
- Reminder: these are C-style arrays

UTILIZING AN ARRAY

BUT FIRST, TERMINOLOGY

- Array, from https://en.cppreference.com/w/cpp/language/array
  - A declaration of the form `T a[N];`, declares 'a' as an array object that consists of 'N' contiguously allocated objects of type 'T'. The elements of an array are numbered 0, …, N - 1, and may be accessed with the subscript operator [], as in a[0], …, a[N - 1].
- Element
  - An individual value within an array
- Index
  - An integer >= 0 that indicates the address of a specific element within an array

ONE MORE TERM

- Capacity
  - The maximum number of values an array can hold
  - Distinct from size (size and capacity have separate meanings in most C++ code)
- What is the maximum index number of any array?
  - Capacity – 1
  - C++ counting begins at 0

AN EXAMPLE

- Consider the following visual representation of an array

20	**50**	**30**	**10**	**40**

- Its capacity is 5
- The 1st element is at index 0 with a value of 20
- …
- The 5th element is at index 4 with a value 40

- We need to understand an array's capacity, allowable range of indices, and element values for arrays to make sense
- Most confusion concerning arrays comes from mixing up these terms and ideas

MORE ARRAY INITIALIZATION

- It is common to want to initialize an array with 0’s
  - There is a shorthand to do this
- `int arr[10] = {0};`
  - 0.0 works for doubles
- What about non-zero initialization?
  - Use a loop
    - For now, there are better ways we’ll see later in the semester

ARRAY INITIALIZATION WITH A LOOP

```
const int CAP = 500;
double hist[CAP];

// array initialization
for (int i = 0; i < CAP; ++i) {
    hist[i] = 1.0;
}
```

A VERY COMMON MISTAKE

- Out of bounds access and/or manipulation
  - Reading or changing memory that is outside of the array
- Typically done by "off-by-one" error in code
- Can also occur via implicit signed to unsigned integer conversion
  - Not as common in Intro, but has happened
- Compiler does NOT catch this
- It is the programmer's responsibility to ensure behavior is correct

SOME SIMPLE PREVENTATIVE MEASURES

- Use a constant to define array capacity; always use constant to refer to the array capacity
- Wherever your array goes, the capacity goes with it
  - When passing array to a function
  - Best practice
- Use C++11 range-based for loop

```
int arr[5] = {1, 2, 3, 4, 5};
for (auto i : arr) {
    std::cout << i << '\n';
}
```

CONST

DIDN'T WE COVER THIS?

- `const`, `const` never changes
  - But we see new use-cases
- Function parameters can be marked `const`
- We tell the compiler to enforce an argument as a read-only value
  - If our intention is just a read a value, the parameter should be marked `const`

CONSIDER THE FOLLOWING

```
void double_array(int a[], int arrSize)
{
    for (int i = 0; i < arrSize; ++i) {
        a[i] *= 2;
    }
}
```

SO FAR, IT'S THE SAME?

- Making a function parameter `const` asks the compiler to enforce the `const`-ness
- Say we a have a function foo with a `const` array parameter
- In foo, I then need to pass the array parameter to function bar, which does not have `const` parameters
- The code will not compile
  - Function foo cannot guarantee the `const`-ness of a parameter when passed to function bar
  - This is because of the "like a reference" way that arrays are passed to functions

SEEMS A HASSLE

- Adding it in after the fact can be
- But compiler-enforced rules tend to make your life easier
  - Much better to catch a `const` issue at compile time, than in production
- Make the decision to be `const` consistent from the beginning
  - It is a best practice, and expected in industry