Arrays Episode I: The ”Phantom” Variables

CS 211

Adam Sweeney

September 30, 2017

Wichita State University, EECS

Introduction

- Consider pieces of information that are closely related
- We have a way of grouping this information

Agenda

- Chapter 7.1
- What is Even an Array
- Arrays in Memory
- Declaring Arrays (& Initializing)
- Utilizing an Array

What is Even an Array

What is Even an Array

- A mechanism to hold and access many variables of the same type through a single name
- Very useful for collecting data
- Grades, scientific readings over time
    - Data that is not just all of the same type
    - Also closely related
    - We could just have a lot of independent variables
    - But we tyically need to operate on the whole set of data
    - This would be extremely tedious

Arrays in Memory

Arrays in Memory

- We should know by know how much space the common simple data types occupy

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

- It makes sense that if we're storing *x* variables, the space occupied in memory is *x * (size of x)*
- What makes arrays special is that the memory dedicated to an array is always contiguous

Declaring & Initializing Arrays

Declaring Arrays

- An array is still a variables
- Our standard rules apply
    - The first part of the declaration is the data type
    - The second part is the name
    - But for arrays, there is now a third part
    - [ ]
- Below is an example of a declaration of an array

```
double grades[49];
```

- This declares an array of doubles that can hold up to 49 values

A Note on Declaring Arrays

- There always needs to be a size
- The size cannot be a user input value (size can not be defined at run-time)
- The size must be known at compile-time
- The size can **not** be changed later

Initializing Arrays

- Like any other variable, an array can be initialized
- But it may take some work
    - We are initializing many variables, after all
- Like any other variable, we can NOT know what our array contains unless we initialize it
- Below is a simple example

```c
char charArr[3] = {'a', 'b', 'c'};
```

- Note, we can initialize all of the elements at once
- Note also, it can be tedious
- We'll look at another way to initialize an array in the next section

Utilizing an Array

Utilizing an Array

- We need to understand some terminology and properties of arrays before we can move on
- What is an array?
    - A collection of variables of the same data type, held under a common name
- What is the size of an array?
    - The maximum number of variables that an array can hold
- Element: A term that denotes an individual variable within an array
- Index: An unsigned integer that indicates the address of a specific element within an array
- What is the maximum index number of an array?
    - $size - 1$

Indexes & Elements

- Consider the following 5-element array

1	2	3	4	5

- Its *size* is 5
- The first element's index is 0, and its value is 1
- The second element's index is 1, and its value is 2
- The third element's index is 2, and its value is 3
- The fourth element's index is 3, and its value is 4
- The fifth element's index is 4, and its value is 5

- We **need** to make the distinction between an array's size, the range of the index values, and the values that are held in each element
- If we confuse these with each other, arrays will not make sense

Initialize an Array with a Loop

- If we have a large array, using $\{ , , , \}$ can be extremely tedious
- Most of the time, we want to initialize arrays all to the same value, or to a pattern-based value
- We can loop that

Example of Array Initialization with a Loop

- Below shows an example of initialzing every element of a large array to zero

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

for (int i = 0; i < SIZE; i++)
    hist[i] = 0;
```

- Our loop begins at zero, because the index of the first element is zero
- We only go until `i < SIZE`, and **never** `i <= SIZE` because the index of the last element is *always* arraySize - 1 (Because we start counting at 0, not 1)

A Very Common and Very Dangerous Mistake

- Out of bounds index access and manipulation is extremely common
- Out of bounds index manipulation is extremely dangerous
- So, what does it mean to go “out of bounds?”
- Recall, an array occupies a contiguous spot in memory
- Out of bounds access means we try to access memory using an index that is not in our array
- The compiler will **not** catch this
- **WE** are responsible for our array access

Consequences

- Out of bounds access
    - Our program will behave unexectedly
- Out of bounds manipulation
    - We may crash the entire system
- When we go out of bounds of our array, there is no telling what memory we are intruding on
- It might belong to our program, it might not
- It might belong to a critical system process, it might not
- With access, we're at least just reading something we shouldn't
- with manipulation, we can possibly do real damage to our system

A Couple Preventative Measures

- Use a constant to define our array size
- Wherever your array goes, the size goes with it
    - This is not required, but it is a best practice
- Use the C++11 standard when looping

```
int intArr[5];
for (int i : intArr) {
    cout << i << " ";
}
cout << "\n";
```

- We won't be using C++11 in this class
- We need to feel comfortable doing it the manual way (but in later classes, C++11 is fair game)