Arrays Episode II: Attack of the Functions

CS 211

Adam Sweeney

September 30, 2017

Wichita State University, EECS

Introduction

- Using arrays alone only gets us so far
- How do arrays and functions interact?

Agenda

- Chapter 7.2
- Indexed variables as function arguments
- Arrays as function arguments
- `const`

Indexed Variables as Function Arguments

Indexed Variables as Function Arguments

- When we use the [ ] to specify a single element of an array, we have a variable of the array type
- An indexed variable can be used as a function parameter

Example

```cpp
#include <iostream>
int adjustDays(int oldDays);
int main()
{
    const int NUM_EMPLOYEES = 3;
    int vacation[NUM_EMPLOYEES], number;

    std::cout << "Enter current vacation days for employees:\n";
    for (int i = 0; i < NUM_EMPLOYEES; i++)
        std::cin >> vacation[i];
    for (int i = 0; i < NUM_EMPLOYEES; i++)
        vacation[i] = adjustDays(vacation[i]);

    std::cout << "Revised vacation is:\n";
    for (int i = 0; i < NUM_EMPLOYEES; i++)
        std::cout << "Employee " << i + 1 << " vacation days = "
                        << vacation[i] << std::endl;
}
```

Link to Example

- The code in the previous slide simply demonstrates how an array element can be used as a function parameter
- The function was not defined
- The link below is a complete example
- `https://repl.it/LkVc/latest/426347`

Passing Entire Arrays to Functions

Passing Entire Arrays to Functions

- This is a thing, and we can do it
- Just a little bit of extra syntax on our part
- Here is a sample function declaration:

  ```
  void fillUp(int a[], int size);
  ```

- Note the use of the square brackets, this is how we know we are passing an entire array to the function
- Note also that there is no size indicated in the square brackets
- You could put one there, but the compiler will ignore it

Passing Arrays (cont.)

```
void fillUp(int a[], int size);
```

- Let's say we have the following variables:

  ```
  int time[10], arrSize = 10;
  ```

- Here's how we call the function:

  ```
  fillUp(time, arrSize);
  ```

- The function's return type is `void`, that is why I don't have to worry about storing a return value

How Arrays Are Passed

- The short answer is “like a reference”
- The longer answer is “as a pointer to the first element”
- When you pass an array as a function parameter, the address of the first element is given
- This is very important, we never have “just” an array
- We have arrays of integers, arrays of characters, etc. Never “just” an array
- These variables are always of the same size
- So, we know that the second element (index 1) is located in memory one *data type size* away from the very beginning of the array

What Does This Matter?

- Unlike passing by value, and more like passing by reference, the array can be directly manipulated
- What if we are passing an array to a function, but want to make sure that it is not altered?

`const`

`const`

- This is not our first time seeing `const`, and it will not be our last
- What `const` does never changes, but we will keep seeing it implemented in new ways
- When we pass arguments to our functions, and they will not be altered, we should make them `const`

An Example

- Consider the following function:

```
double calculateAverage(int a[], int arrSize)
{
    int sum = 0;
    for (int i = 0; i < arrSize; i++)
        sum += a[i];
    return static_cast<double>(sum) / arrSize;
}
```

- We are not altering the array
- This is a very simple example; in a more complicated function there is the very real possibility that we can accidentally manipulate an element when we don't want to

Using `const` as a Safety

- If we instead define the function like so:

```
double calculateAverage(const int a[], const int arrSize)
{
    int sum = 0;
    for (int i = 0; i < arrSize; i++)
        sum += a[i];
    return static_cast<double>(sum) / arrSize;
}
```

- Now we have guaranteed that neither argument will be modified
- The compiler will enforce this for us

About that Enforcement

- In order to guarantee `const` correctness for function parameters, you need to go all in or not at all
- `const` will enforce its read-only attribute all the way down a chain
- A chain in our case will be a function that calls another function
- If we are passing a `const` parameter to another function, it **also** needs to be declared as a `const` parameter in the other function as well

const Enforcement Example (This fails to compile)

```
// Processes array to find average of the element values
double computeAverage(int a[], int size);

// Will show how each element differs from the average
void showDifference(const int a[], int size)
{
    double average = computeAverage(a, size);
    cout << "Average of the " << size
        << " numbers = " << average << endl
        << "The numbers are:\n";
    for (int index = 0; index < size; index++)
        cout << a[index] << " differs from average by "
            << (a[index] - average) << endl;
}
```

What's Going On?

- In the `showDifference()` function, the array is declared as `const`
- When we give this `const` array to the `computeAverage()` function, the array parameter of `computeAverage()` is not `const`
- What this means, is that `showDifference()` can no longer guarantee that its array remains `const`
- Because of this, we get a compile error
- `https://repl.it/LmsF/latest/426347`

Seems a Hassle

- If you are attempting to go back and make code `const` consistent, it will be
- Using `const` is something you go all or nothing on, and that decision should be made before you write any code.
- using `const` is recommended