Arrays

Shaan Mathur

November 13, 2018

Often times, the requirements of the software we build ask us to be able to handle large swamps of data. But if all we had were just variables, its not obvious how we could elegantly be able to deal with a large quantity of data. For instance, suppose we expect as input the daily average temperature of Westwood for over the last 8 years. Surely we don't want to have 2922 different integer variables...

Listing 1: A Poor Life Without Arrays...

```
// Okay
int temp1;
cin >> temp1;

// Okay.
int temp2;
cin >> temp2;

// Okay...
int temp3;
cin >> temp3;

/*
  ...
*/

// Why
int temp2922;
cin >> temp2922;
```

And what if we didn't even know until runtime how many temperatures we'd be receiving (maybe the first integer input by the user is the number of temperatures that will be provided, followed by those temperatures)? Obviously we need a better approach.

To that end, we introduce the notion of an *array*, a contiguous (yes, I said contiguous) block of memory that can store many variables. For instance, we might declare an integer array named *arr* of length 2922, where we can refer to the i^{th} integer by *arr[i]* (and we start counting from $i = 0$). So instead we could change our program from earlier to something like this:

Listing 2: A Life With Arrays

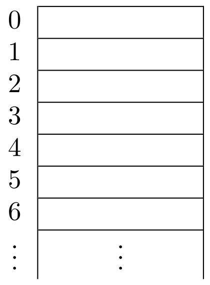
```
int arr[2922];
for (int i = 0; i < 2922; i++) {
    cin >> arr[i];
}
```

Now *that* is a lot more elegant.

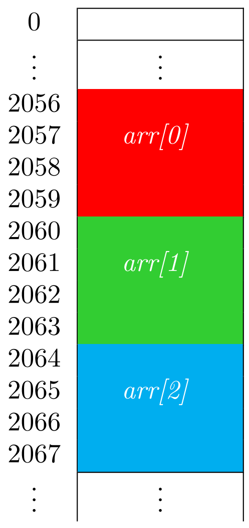
1 How Memory Works (Optional But Recommended)

Break open your computer and you'll see a bunch of sticks of memory connected to your hardware in miscellaneous locations. The operating system and hardware have to work together to be able to actually figure out where to store your program's variables and data, since everything is all over the place! Luckily as programmers, we don't have to deal with such a headache, due to the grand illusion known as *virtual memory.* Although *virtual memory* is mostly beyond the scope of this course (but it won't be for CS33 and CS111!), understanding it can be very helpful.

Even though you and I both know memory in our computer is *all over the place* and, most importantly, is *finite*, we can pretend otherwise. We can consider memory as one infinitely long, contiguous block of memory.



Now when you declare an array of 3 integers, a chunk of 12 bytes (4 bytes per integer) is reserved for your usage.



So when you receive a contiguous chunk of memory, it is a contiguous chunk in this infinitely long virtual memory that you are actually being provided. Although virtual memory is an illusion, arrays tend to be actually contiguous in physical memory since this can lead to performance boosts (if interested, check out the topic of caching and spatial/temporal locality).

2 Using C++ Arrays

2.1 Declaring and Initializing Arrays

Declaring arrays is as simple as declaring normal variables.

`int age;` vs. `int ages[100];`

The only difference is the square brackets and length. However, there is an important caveat: **the length must be a constant**. Examine the following examples.

Listing 3: Fixed Length Arrays

```cpp
int length = 5;
const int LENGTH = 5;

int temperatures[5];
double scores[LENGTH];

bool isVisited[length]; // Compile error!
```

Using an integer literal (so long as it is ≥ 0) for array length is fine, since the length is predetermined (and is hence constant); using an integer with the `const` specifier makes a promise to the compiler that the value will not change, so it is also fine; however using just an integer with no `const` specifier is not okay for declaring array lengths, since the integer *could* change at some point in the program (even if it obviously won't in this simple program).

We can initialize arrays directly in different ways, or we could initialize the elements separately.

Listing 4: Array Initialization

```
int scores[5] = {98, 84, 75, 93, 96}; // Length is 5
double costs[] = {54.30, 53.25, 73.41}; // Length is 3
int magic[5] = { }; // Sets all elements to 0.
int notmagic[5];    // Each entry has a garbage, undefined value

char str[3];
str[0] = 'h';
str[1] = 'i';
str[2] = '\0'; // str is {'h', 'i', '\0'}
```

2.2 Arrays of Arrays

Sometimes the structure of some problems are more inclined to the idea of arrays with multiple dimensions. For instance, a video game with a maze in it might want a 2-dimensional grid representation, where perhaps an 'X' denotes a wall and a '.' denotes a free space.

X	X	X	X	X
X	.	X	X	X
X	.	.	.	X
X	.	X	.	X
X	.	.	.	X
X	X	X	X	X

Maybe we could introduce a new notion of a *2-dimensional array* where instead of just having one index, we could have two (e.g. *maze[row][column]*). However rather than introducing a new structure altogether, we can instead have an array of arrays; more specifically, we can represent an $n \times m$ grid by having an array (of length n) of arrays (each of length m). The syntax is markedly similar to how single dimensional arrays would work.

`double scores[30];` vs. `char maze[20][30];`

Direct initialization of a two-dimensional array is perhaps more enlightening in terms of the actual structure of a two-dimensional array.

Listing 5: 2D Maze

```
char maze[6][5] =
    {
        {'X', 'X', 'X', 'X', 'X'}, // maze[0]
        {'X', '.', 'X', 'X', 'X'}, // maze[1]
        {'X', '.', '.', '.', 'X'}, // maze[2]
        {'X', '.', 'X', '.', 'X'}, // maze[3]
        {'X', '.', '.', '.', 'X'}, // maze[4]
        {'X', 'X', 'X', 'X', 'X'}, // maze[5]
    };
// maze[1][0] == 'X';
// maze[4][1] == '.';
// maze[4][4] == 'X';
```

Thus when we say *maze[i][j]*, we are really first asking for the i^{th} array in *maze*, and then asking for the j^{th} character in *maze[i]*. More generally, we can have multidimensional arrays of any finite dimension, where a 3D array is an array of 2D arrays, a 4D array is an array of 3D arrays, and so on. Understanding mult-dimensional arrays from this viewpoint makes it simpler to understand other C++ rules for multidimensional arrays that are just consequences of rules for arrays.

Listing 6: Multidimensional Array Initialization

```
int arr2D[2][3] = { {1, 2, 3}, {4, 5, 6} };

// 1D arrays can specify type (int) without length
int arr[] = {98, 99, 100};

// 2D arrays can specify type (int arrays of length 3)
// without length
int matrix[][3] = { {2, 4, 8}, {3, 9, 27}, {4, 16, 64} };

// This is a compile error since type isn't fully specified
// (int arrays of length 3)
int matrix[][] = { {2, 4, 8}, {3, 9, 27}, {4, 16, 64} };

/*  Multidimensional arrays can omit only leftmost length
 *  all others needed to specify type (e.g. array of
 * length 4 of arrays of length 3 of doubles)
 */
double coord[][4][3] = {
    {
        {36.2, 13.4, 2.1},
        {41.2, 1.0, -14.4},
        {1.9, 11.2, -44.4},
        {11.6, 1.3, 94.4}
    },

    {
        {3.62, .33, 24.1},
        {1.2, 13.0, 144.4},
        {81.9, 31.2, -44.4},
        {24.6, 15.3, 54.1}
    }
};

// Initializes everything to 0
int zeroes[50][199][131] = { { { } } };
```

2.3 Arrays as Parameters

When you call some function *foo* with function signature

`void foo(int x);`

what happens? Suppose the code invokes *foo* through a call like *foo(y)*, where y is some integer variable. It turns out that when the code for *foo* is run, x is given the value of y; more specifically, the value of y is copied into local variable x.

Now imagine we want to write a function that finds the maximum score in an array of over a million integers. If we pass an array to this function, should we copy the **entire 4 million bytes** into this function's parameter? That would be absurd, especially if we don't even plan to change (mutate) the array. If the array is somewhere in memory already, why not just make the function's parameter refer to that directly instead of giving the parameter its own unique copy?

To that end, arrays as arguments to functions are always *passed by reference* in the sense that the function's array *is* the array the caller passed to it. If you change that array, you change the caller's array.

Listing 7: Arrays as Arguments

```
// first length of array is omitted, all others are required. Why?
void foo(int arr[][10][13], int length) {
    // Do something
}
```

Listing 8: Arrays are Passed by Reference

```
void increment(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        ++arg[i];
    }
}

int main() {
    // Usually good practice to define arbitrary constants
    const int LENGTH = 6;
    int temperatures[LENGTH] = {80, 75, 75, 85, 90, 86};
    increment(temperatures, LENGTH);
    // temperatures is now {81, 76, 76, 86, 91, 87}
}
```