Chapter

19

Introducing Arrays

C++ has a built-in array type, inherited from the C language. The array builds upon the pointer concepts you covered in the last few lessons. Arrays (unlike `vector`s) are directly supported by your hardware.

Arrays are similar to the `vector` library type, except:

- Arrays are allocated with a fixed size that you can't subsequently change.
- The size is not stored along with the array.
- Arrays offer no support for inserting and deleting elements.
- C++ performs no bounds-checking on arrays.
- Array elements may be allocated on the stack or static area (unlike `vector`s).

Arrays are the low-level plumbing from which the more powerful collection classes are built. To understand the implementation of those classes, you need to have some familiarity with the mechanics of arrays.

Defining Arrays

An array must be defined before it is used:

```
base-type name[size];
```

The definition requires a base type, name, and allocated size; size is a positive integer constant expression indicating the number of elements. For example:

```
const size_t SIZE{10};
int a[SIZE];
```

This creates an array of `10` elements, each of which is of type `int`.

- Specify the size as a symbolic constant instead of a literal, so the array size is easier to change in your code.
- The size must be positive; zero or negative are illegal.
- The size must be constant; a regular, non-`const` variable should not work, although some compilers may permit it.

- If defined inside a function, the **elements are on the stack**; if defined outside of a function, the elements are allocated in the **static storage area**.

Index numbers begin with **0** and run up to the **array size minus one**.

> *This is different than Java where the array variable and the allocated actual array are different. In C++ there is no array variable equivalent.*

Array Initialization

When you define an array, **its elements are default initialized**:

- If the base type is a primitive and the array is **local**, they are **uninitialized**. Note: this is **unlike** `vector` where they are initialized to **0**.
- If the base type is a primitive and the **array is global** or `static`, then the elements are set to **0**.
- If the base type is a **class type** then the **default constructor** for the type is used to initialize each element. (This is different than Java or C#.)

Arrays **may be explicitly initialized** at definition time using a list of initializers enclosed in curly braces. C++11 list initialization was patterned after this **built-in array feature**. Note, however, with the array you must add the = sign.

```
const int DIGITS[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

When using an **explicit initializer** you may skip the array size; the compiler counts the number of values you supply. With **DIGITS** the compiler **implicitly** supplies the size **10**.

You can **combine these two forms of definition**; you can specify an array allocation size (or **dimension**) **and** provide an initializer list as well.

- If you have **fewer initializers** than the size, the others **are set to zero**.
- If you supply an allocated size, then the number of initializers **cannot exceed** the dimension.

```
const size_t LEN{3};
int a1[LEN] = {0, 1, 2};          // [0, 1, 2]
int a2[] = {0, 1, 2};             // [0, 1, 2]
int a3[LEN + 2] = {0, 1, 2}       // [0, 1, 2, 0, 0]
int a4[LEN - 1] = {0, 1, 2}       // ERROR
int a5[LEN];                      // Uninitialized
int a6[LEN] = {};                 // [0, 0, 0]
```

The Array Size

Suppose that you have an array containing the names of all U.S. cities with populations of over **1,000,000**. Taking data from the 2010 census, you write:

```
const string CITIES[] = {
    "New York", "Los Angeles", "Chicago",
    "Houston", "Philadelphia", "Phoenix",
    "San Antonio", "San Diego", "Dallas",
};
```

This **changes over time**. In 2020, both San Jose and Austin Texas joined the list. Fortunately, you **may** simply **add new names to the list**, or delete them, and **let the compiler count how many there are**. This is so common, **you are allowed to leave a trailing comma** (such as the one after Dallas) and **it doesn't create a syntax error**.

The Allocated Size

So, **how do you know how many cities there are?** You don't want **to have to count** them! After all, **the compiler knows** how many there are. C++ has a **standard idiom** for determining the **allocated size** of an array **at compile time**, provided **the array definition is in scope**.

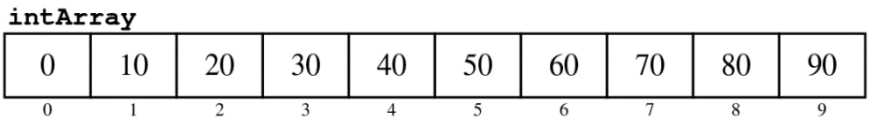
```
constexpr auto LEN{sizeof(CITIES) / sizeof(CITIES[0])};
```

This compile-time expression takes the size of the entire array (returned from the **sizeof** operator, and thus of type **size_t**), and divides it by the size of the initial element in the array. Because all elements of an array are the same size, the result is the number of elements in the array, regardless of the element type.

Array Selection

Array selection uses the **subscript operator** as in Java. The result of selection expression is an ***lvalue***, which means that you can assign new values to it, like this:

```
for (size_t i = 0; i < LEN; ++i) { intArray[i] = 10 * i; }
```



C++ **performs no bounds checking** on array selection. Unlike **vector**, **there is no safe, range-checked alternative**, such as **at()**. If the array index is out of range, your compiler decides where the element **would be** in memory, leading to **undefined results**.

Worse still, if you assign a value to that index position, you **can overwrite** the contents of memory used by some other part of the program. Writing beyond the end of an array is one of the primary vulnerabilities used to attack computer systems.

Arrays and Loops

Just like **vectors**, the real advantage of arrays is that you can automate the processing of a collection of related elements, like the grades for all the students in a class. However, because arrays are lower-level structures, processing them is not quite as convenient.

Counter-Controlled Loops

Most commonly, you'll use a counter-controlled loop, like this:

```
const int LEN = 5;
int a[LEN];
for(int i = 0; i < LEN; ++i)
{
   a[i] = (i + 1) * 2;
}
```

Unlike **string** and **vector**, arrays have no built-in **size()** member function. That means, for a counter-controlled loop, the array's **length** must be stored in a variable prior to entering the loop. The array's length **is a constant.** Here I've used an **int**, but you may use a **size_t** as with **vector**, as well.

Other Loops

You may use the **range-based *for*** loop on arrays, provided that the array definition is **in scope**. Here's an example:

```
int a[] = {...};
for (auto e : a)
{
   cout << e << " ";
}
```

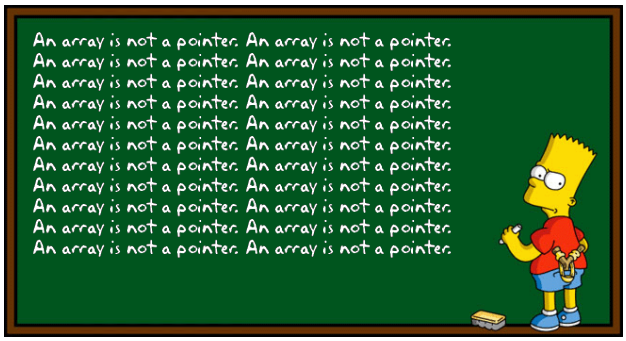
Similarly, you can use an **iterator loop**, using the **begin()** and **end()** functions which you met with **vector**. Finally, you can use a **sentinel loop** provided you place a special value in the array to mark its end. This is used with character arrays, as you'll see when we cover C-style strings.

Pointers & Arrays

Because an array name is the same as the address of its first element, you can use it as **a pointer value**. The crucial **difference between arrays and pointers** in C++ comes into play when variables are declared.

```
int array[5];
int *p;
```

The distinction between these is **memory allocation**. The first reserves five **int** values; the second **reserves space for a pointer**. The name **array** is an address, not a pointer.



If you define an array, **you have storage to work with**; if you declare a pointer variable, that variable **is not associated with any storage** until you initialize it. The simplest way to initialize a pointer to an array is to **assign the array name to the pointer variable**:

```
int array[5];
int *p = array;
```

Now, the pointer **p** contains the same address used for **array**.

Pointer Arithmetic and Arrays

You can **change the location** where a pointer points by incrementing or decrementing it. You can also **generate new pointer** values by adding or subtracting integers from an existing pointer. The **effective address** depends upon the **base type** of the pointer.

If you add 1 to an integer address, the new address produced is 4 bytes larger than the original pointer value; if you add 1 to a double address, the new address is 8 bytes larger.

```
int a[10];
int *p = a;      // p <- &a[0]
p = a + 1;       // p <- &a[1]
p = a + 5;       // p <- &a[5]
```

Of course, unlike a pointer, you **cannot increment or decrement** an array name.

You can also **dereference an array**, just like a regular pointer, using the regular **dereferencing operator**. Here are some examples (note the parentheses):

```
int a[10];
cout << *a;          // a[0]
cout << *(a + 2);    // a[2]
cout << *a + 2;      // a[0] + 2
```

Similarly, you can **combine dereferencing with address arithmetic**, using the **subscript operator**; this works for both pointers and arrays. All of these expressions are true.

```
int a[10];
int *p = a;
*a == *p;            // true
a[0] == p[0];        // also true
a[3] = *(p + 3);     // true again
4[a] = *(4 + a);     // also true
```

In fact, in C++, the subscript operator is just shorthand for a combination of address arithmetic along with dereferencing the resulting address.

address[offset] -> offset[address] -> *(address + offset)

Array Characteristics

The **array name** is synonymous with the **address of its first element**. This address cannot be changed; **it is constant**. Look at this code fragment:

```
int list[5];
cout << list << endl;
```

What prints is the **address of the first element**; not the contents of the first **int**, and not the entire array. Here's one possible output. Click the link above to try it out.

```
0x7a638a937360
```

Assignment and Comparison

An array name **is not a variable**. You **cannot assign** to an array name, nor **meaningfully compare** two arrays using the built-in comparison operators.

```
int a1[] = {1, 2, 3}, a2[3];
a2 = a1;                    // 1. Illegal
a2[0] = a1[0];              // 2. Fine
a2 = {1, 2, 3};             // 3. Illegal
if (a1 == a2) . . .         // 4. Legal, but stupid
```

The arrays **a1** and **a2** are **the same type and dimension**. Given that:

1. It is **illegal to assign a1 to a2**. The name **a1** is the constant address of the first element in the array. **It is not a variable** that can be assigned to. This is completely different from structures, where **a1** and **a2** **would** both be variables (**lvalues**) and thus **could be** assigned to.
2. It **is legal** to **assign to array elements**; **a1** **is not** a variable, but **a1[0]** **is**.
3. **You can list initialize** an array, but you **cannot list assign** to the array.
4. With structures, **comparing two variables** is a syntax error. **Comparing two array names, while legal, is very stupid**.
5. Since the array name is the address of the first element in the array, and since the two arrays **cannot live at the same location** in memory, the comparison must be **false**. It doesn't matter what is inside the array.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **make submit**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.