Introducing Arrays

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

Arrays are similar to the `vector` library type, except that they are much simpler, and have many fewer features:

- Arrays are **allocated with a fixed size** when created, which cannot change.
- The array size **is not stored along with the array** data elements.
- Arrays have **no built-in functions**, such as support for **inserting** or **deleting**.
- C++ performs **no bounds-checking** on arrays.
- Array elements may be allocated on the stack or static area so they are **more efficient** than `vector`, whose elements are always allocated on the heap, even if the `vector` itself is on the stack.

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.

Arrays vs. vector

Why use an array if `vector` is so much more convenient? Why bother learning about arrays at all? Here's why.

- **`vector` elements** are always **allocated on the heap**
- Arrays may be **allocated on the stack**, **static area**, **or the heap**. This avoids performance issues that arise with dynamic memory.
- Arrays often have **higher performance** and **take less memory**.
- Arrays are generally used for low-level **systems programming** (operating systems)
- Array performance is **deterministic**; for this reason, arrays are normally used for **embedded programs** that must run for long periods of time.

In short, arrays are usually faster and may take less memory than dynamic library types like **`string`** and **`vector`**. Using arrays in C++ is programming **as the CPU sees it**.

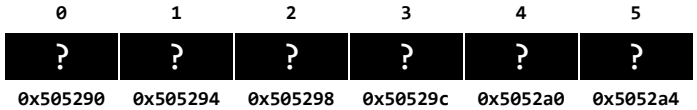
Defining Arrays

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 the compiler to allocated. For example:

```
const size_t kSize = 6;
int a[kSize];
```



This creates an array named `a`, of `6` elements, each of which is an **uninitialized** `int`.

- A good practice is to **specify the size as a symbolic constant** instead of a literal.
- 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**.

*C++ arrays are different than those in Java where the array variable and the allocated actual array are different. In C++ there is no array variable equivalent. Instead, the array name (`a` in the example) is an alias for the **address of the first element**, `0x505290` here.*

Array Initialization

When you define an array, its elements are default initialized. So, what does that mean?

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

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 explicit 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 a size, then the number of initializers **cannot exceed the dimension.**

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

The Allocated 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 would write:

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

However, the size of the cities **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 then **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**.

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 size_t kCitiesSize = 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:

```
const int kLen = 6;
int a[klen];
for (size_t i = 0; i < kLen; ++i) {
    a[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, and performs the requested operation, 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.

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 contents of the entire array. Here's one possible output. Click the link to try it.

```
0x505260
```

Since an array name **is not a variable**, you **cannot assign** to an array, nor, can you **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 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**.

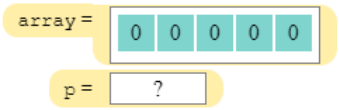
Since the array name is the address of the first element in the array, and since the two arrays **cannot live at the same location** in memory, the comparison must be `false`. It doesn't matter what is inside the array.

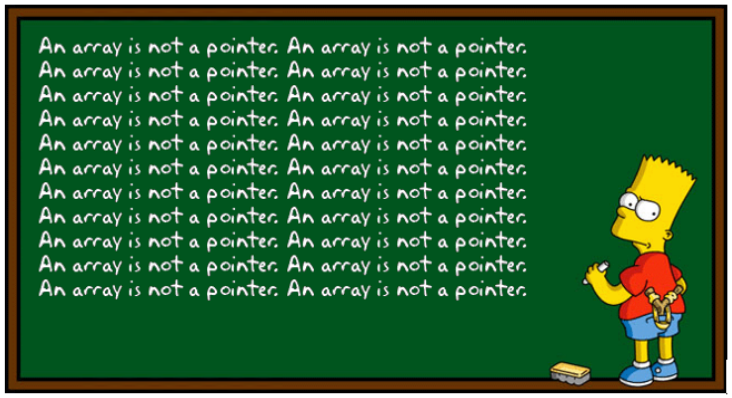
Pointers & Arrays

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.

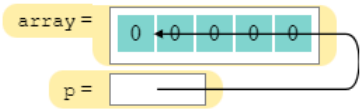
```
int array[5] = {0};
int *p;
```

The distinction between these is **memory allocation**. The first reserves five **int** values (on the stack or in the static storage area); 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] = {0};
int *p = array;
```

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

Pointer Arithmetic & 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.

Given `4`-byte `int` and `8`-byte `double`, 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 array[] = {1, 2, 3, 4, 5};
int *p = array;     // p <- &array[0]
p = array + 1;      // p <- &array[1]
p = array + 4;      // p <- &array[4]
```

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

Dereferencing Arrays

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

```
int array[] = {1, 2, 3, 4, 5};
cout << *array;          // array[0]
cout << *(array + 2);    // array[2]
cout << *array + 2;      // array[0] + 2
```

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

```
int array[] = {1, 2, 3, 4, 5};
int *p = array;
*array == *p;              // true
array[0] == p[0];          // also true
array[3] = *(p + 3);       // true again
4[array] = *(4 + array);   // 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)
```

Arrays & Functions

You cannot pass an array by value to a function as you can a vector. While you can pass an array to a function by reference, you'll almost never do that. Instead, we need to learn about a new way of passing parameters: **pass by address**.

Recall that an array name **is an address**, which you may store inside a pointer.

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

This is the secret to writing functions that process arrays:

- Create a function **with a pointer as a parameter**. You **may** declare this pointer as `int a[]`, indicating that you **intend** to initialize it with the address of the first element in an array.
- **Call the function**, **supplying the name of an array** as the argument.

Here are two prototypes. The first uses the square brackets to declare the pointer variable `a`. The second uses the normal pointer parameter syntax. Both have identical meaning **when as a parameter declaration**.

```
int aSum(const int a[], size_t size);
int aSum(const int *a, size_t size);
```

With "pointer notation", the star comes **before** the name, while with "array notation", the **brackets come after**. A common error, for Java programmers moving to C++, is to write the prototype like this,

```
int aSum(const int[] a, size_t size);
```

which is a syntax error.

Decaying Arrays

When you pass an array to a function, we say that the array "decays to a pointer". This is similar to what happens with primitive types in this case:

```
int n = 3.14;
```

The `int` variable `n` cannot store the fractional portion of `3.14`, so it **truncates the number** and stores `3`.

When you pass an array name to a function, and it is **converted into a pointer**, it also **loses certain information**; specifically, it **loses the ability to determine the allocated size of the array** inside the function.

When you **declare** the array, the compiler "knows" the allocated size of the array:

```
int array[] = {...};
size_t kLen = sizeof(array) / sizeof(array[0]);   // OK;
```

However, you pass that array to a function, you **cannot** use the same code.

```
void f(const int a[]) {
  size_t kBug = sizeof(a) / sizeof(a[0]);   // ERROR
}
```

That means we **must calculate an array size when the array is created**, and then **supply it when calling the function**.

Arrays & Const

Arrays passed to a function act as if the array was passed by reference. That can be dangerous, because the function may inadvertently modify the caller's argument.

```
1  for (size_t i = 0; i < len; ++i)
2  {
3      sum += a[i];
4      a[i] = sum;
5  }
```

This function was **intended to sum all the elements** in an array. If you were distracted and inadvertently used an assignment operator instead of the comparison operator, as on line 4, the function would still produce the correct sum, but mistakenly **destroy the values in the array passed to it**.

Not a good thing. To fix this, use the same technique you used for pass-by-reference:

- If a function **intends** to modify the array (initialization, shifting, sorting, etc.) then **do not use** `const` in front of the formal parameter. (Since you are passing by address, **you will never use** `&`.)
- If a function **does not intend to modify the array** (counting, summing, printing, etc.) then **always** use `const` in front of the parameter.

```
double average(const int a[], size_t len);
```

Arrays and Loops

Just as with `vector`, 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. There are several ways to use loops to traverse an array.

1. Calculate the **number of elements in the array** and use that as a limit on a traditional counter-controlled ***`for`*** or ***`while`*** loop.
2. Use a **sentinel value** stored in the array to mark its end.
3. Use a pair of pointers: one to the first element in the array, and one to the address right past the end of the array. These are called **iterator-based** loops.
4. Use the C++ 11 **range-based** ***`for`*** loop.

Inside a function, **only the first three are meaningful**. You **cannot** use the range-based for loop on an array name after it has decayed to a pointer.

The range-based for Loop

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

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

Counter-Controlled Loops

Inside a function, most commonly, you'll calculate the size of an array at the point where it is declared, and then **pass that size as an additional argument** when you **call** the function.

For instance, here is a function which sums the elements in a **`vector`**:

```
int sum(const vector<int>& v)
{
   int sum = 0;
   for (size_t i = 0, len = v.size(); i < len; ++i)
      sum += v.at(i);
   return sum;
}
```

Notice that the function only requires one argument, since the **`vector`** "carries" its size along with it. With an array, you'd need to write the same function like this:

```
int sum(const int array[], size_t len)
{
   int sum = 0;
   for (size_t i = 0; i < len; ++i)
      sum += array[i];
   return sum;
}
```

Unlike **`string`** and **`vector`**, arrays have no built-in **`size()`** member function. And, because **`array`** is really a pointer, you can't use the **`sizeof`** "trick" inside the function. You must pass the length as an argument **when calling** the function. Note, also, that unlike **`vector`** you have no range-checked **`at()`** function. You must use the built-in subscript operator.

Sentinel Loops

If an array contains a special value marking the end of the data, then you can use a **sentinel loop** to process the array. In the long-ago data-processing days, for instance, an array of positive values would contain a `-1` as a terminator or sentinel.

One **advantage** of the sentinel technique, is that it allows you to write functions that process variable amounts of data. (Later, we'll look at **partially-filled arrays**, which are a more general solution to this problem.)

Here's an example using a sentinel loop, which calculates the average grade on a test:

```
double average(const double grades[])
{
   double sum = 0;
   int count = 0;
   for (size_t i = 0; grades[i] >= 0; ++i)
   {
      sum += grades[i];
      count++;
   }
   return sum / count;
}
```

The **disadvantage of this technique** is that your function needs to trust that the array has been properly terminated, which is really a security hole. This technique is not used that widely for numeric arrays. However, C-style strings use a special sentinel (called the NUL byte) to mark the end of the string, and all C-string functions employ sentinel loops.