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]
```