17.7 — Introduction to C-style arrays

Now that we've covered `std::vector` and `std::array`, we'll complete our coverage of arrays by covering the last array type: C-style arrays.

As mentioned in lesson 16.1 -- Introduction to containers and arrays, C-style arrays were inherited from the C language, and are built-in to the core language of C++ (unlike the rest of the array types, which are standard library container classes). This means we don't need to #include a header file to use them.

As an aside…

Because they are the only array type natively supported by the language, the standard library array container types (e.g. `std::array` and `std::vector`) are typically implemented using a C-style array.

Declaring a C-style array

Because they are part of the core language, C-style arrays have their own special declaration syntax. In an C-style array declaration, we use square brackets (`[]`) to tell the compiler that a declared object is a C-style array. Inside the square brackets, we can optionally provide the length of the array, which is an integral value of type `std::size_t` that tells the compiler how many elements are in the array.

The following definition creates a C-style array variable named `testScore` which contains 30 elements of type `int`:

```
int main()
{
    int testScore[30] {};      // Defines a C-style array named testScore that
contains 30 value-initialized int elements (no include required)

//  std::array<int, 30> arr{}; // For comparison, here's a std::array of 30 value-
initialized int elements (requires #including <array>)

    return 0;
}
```

The length of a C-style array must be at least 1. The compiler will error if the array length is zero, negative, or a non-integral value.

The array length of a c-style array must be a constant expression

Just like `std::array`, when declaring a C-style array, the length of the array must be a constant expression (of type `std::size_t`, though this typically doesn't matter).

Tip

Some compilers may allow creation of arrays with non-constexpr lengths, for compatibility with a C99 feature called variable-length arrays (VLAs).

Variable-length arrays are not valid C++, and should not be used in C++ programs. If your compiler allows these arrays, you probably forgot to disable compiler extensions (see 0.10 -- Configuring your compiler: Compiler extensions).

Subscripting a C-style array

Just like with a `std::array`, C-style arrays can be indexed using the subscript operator (`operator[]`):

```
#include <iostream>

int main()
{
    int arr[5]; // define an array of 5 int values

    arr[1] = 7; // use subscript operator to index array element 1
    std::cout << arr[1]; // prints 7

    return 0;
}
```

Unlike the standard library container classes (which use unsigned indices of type `std::size_t` only), the index of a C-style array can be a value of any integral type (signed or unsigned) or an unscoped enumeration. This means that C-style arrays are not subject to all of the sign conversion indexing issues that the standard library container classes have!

```
#include <iostream>

int main()
{
    const int arr[] { 9, 8, 7, 6, 5 };

    int s { 2 };
    std::cout << arr[s] << '\n'; // okay to use signed index

    unsigned int u { 2 };
    std::cout << arr[u] << '\n'; // okay to use unsigned index

    return 0;
}
```

Tip

C-style arrays will accept signed or unsigned indexes (or unscoped enumerations).

`operator[]` does not do any bounds checking, and passing in an out-of-bounds index will result in undefined behavior.

As an aside…

When declaring an array (e.g. `int arr[5]`), the use of `[]` is part of the declaration syntax, not an invocation of the subscript operator `operator[]`.

Aggregate initialization of C-style arrays

Just like `std::array`, C-style arrays are aggregates, which means they can be initialized using aggregate initialization.

As a quick recap, aggregate initialization allows us to directly initialize the members of aggregates. To do this, we provide an initializer list, which is a brace-enclosed list of comma-separated initialization values.

```
int main()
{
    int fibonnaci[6] = { 0, 1, 1, 2, 3, 5 }; // copy-list initialization using braced
list
    int prime[5] { 2, 3, 5, 7, 11 };         // list initialization using braced list
(preferred)

    return 0;
}
```

Each of these initialization forms initializes the array members in sequence, starting with element 0.

If you do not provide an initializer for a C-style array, the elements will be default initialized. In most cases, this will result in elements being left uninitialized. Because we generally want our elements to be initialized, C-style arrays should be value initialized (using empty braces) when defined with no initializers.

```
int main()
{
    int arr1[5];    // Members default initialized int elements are left
uninitialized)
    int arr2[5] {}; // Members value initialized (int elements are zero
uninitialized) (preferred)

    return 0;
}
```

If more initializers are provided in an initializer list than the defined array length, the compiler will error. If fewer initializers are provided in an initializer list than the defined array length, the remaining elements without initializers are value initialized:

```
int main()
{
    int a[4] { 1, 2, 3, 4, 5 }; // compile error: too many initializers
    int b[4] { 1, 2 };          // arr[2] and arr[3] are value initialized

    return 0;
}
```

One downside of using a C-style array is that the element's type must be explicitly specified. CTAD doesn't work because C-style arrays aren't class templates. And using `auto` to try to deduce the element type of an array from the list of initializers doesn't work either:

```
int main()
{
    auto squares[5] { 1, 4, 9, 16, 25 }; // compile error: can't use type deduction
on C-style arrays

    return 0;
}
```

Omitted length

There's a subtle redundancy in the following array definition. See it?

```
int main()
{
    const int prime[5] { 2, 3, 5, 7, 11 }; // prime has length 5

    return 0;
}
```

We're explicitly telling the compiler the array has length 5, and then we're also initializing it with 5 elements. When we initialize a C-style array with an initializer list, we can omit the length (in the array definition) and let the compiler deduce the length of the array from the number of initializers.

The following array definitions behave identically:

```
int main()
{
    const int prime1[5] { 2, 3, 5, 7, 11 }; // prime1 explicitly defined to have
length 5
    const int prime2[] { 2, 3, 5, 7, 11 };  // prime2 deduced by compiler to have
length 5

    return 0;
}
```

This only works when initializers are explicitly provided for all array members.

```
int main()
{
    int bad[] {}; // error: the compiler will deduce this to be a zero-length array,
which is disallowed!

    return 0;
}
```

When using an initializer list to initialize all elements of a C-style array, it's preferable to omit the length and let the compiler calculate the length of the array. That way, if initializers are added or removed, the length of the array will automatically adjust, and we are not at risk for a mismatch between the defined array length and number of initializers provided.

Best practice

Prefer omitting the length of a C-style array when explicitly initializing every array element with a value.

Const and constexpr C-style arrays

Just like `std::array`, C-style arrays can be `const` or `constexpr`. Just like other const variables, const arrays must be initialized, and the value of the elements cannot be changed afterward.

```
#include <iostream>

namespace ProgramData
{
    constexpr int squares[5] { 1, 4, 9, 16, 25 }; // an array of constexpr int
}

int main()
{
    const int prime[5] { 2, 3, 5, 7, 11 }; // an array of const int
    prime[0] = 17; // compile error: can't change const int

    return 0;
}
```

The sizeof a C-style array

In previous lessons, we used the `sizeof()` operator to get the size of an object or type in bytes. Applied to a C-style array, `sizeof()` returns the number of bytes used by the entire array:

```
#include <iostream>

int main()
{
    const int prime[] { 2, 3, 5, 7, 11 }; // the compiler will deduce prime to have
length 5

    std::cout << sizeof(prime); // prints 20 (assuming 4 byte ints)

    return 0;
}
```

Assuming 4 byte ints, the above program prints `20`. The `prime` array contains 5 `int` elements that are 4 bytes each, so 5 * 4 = 20 bytes.

Note that there is no overhead here. An array object contains its elements and nothing more.

Getting the length of a C-style array

In C++17, we can use `std::size()` (defined in the <iterator> header), which returns the array length as an unsigned integral value (of type `std::size_t`). In C++20, we can also use `std::ssize()`, which returns the array length as a signed integral value (of a large signed integral type, probably `std::ptrdiff_t`).

```
#include <iostream>
#include <iterator> // for std::size and std::ssize

int main()
{
    const int prime[] { 2, 3, 5, 7, 11 };   // the compiler will deduce prime to have
length 5

    std::cout << std::size(prime) << '\n';  // C++17, returns unsigned integral value
5
    std::cout << std::ssize(prime) << '\n'; // C++20, returns signed integral value 5

    return 0;
}
```

Getting the length of a C-style array (C++14 or older)

Prior to C++17, there was no standard library function to get the length of a C-style array.

If you're using C++11 or C++14, you can use this function instead:

```
#include <cstddef> // for std::size_t
#include <iostream>

template <typename T, std::size_t N>
constexpr std::size_t length(const T(&)[N]) noexcept
{
        return N;
}

int main() {

        int array[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
        std::cout << "The array has: " << length(array) << " elements\n";

        return 0;
}
```

This uses a function template that takes a C-style array by reference and then returns the non-type template parameter representing the array's length.

In much older codebases, you may see the length of a C-style array determined by dividing the size of the entire array by the size of an array element:

```
#include <iostream>

int main()
{
    int array[8] {};
    std::cout << "The array has: " << sizeof(array) / sizeof(array[0]) << "
elements\n";

    return 0;
}
```

This prints:

```
The array has: 8 elements
```

How does this work? First, note that the size of the entire array is equal to the array's length multiplied by the size of an element. Put more compactly: `array size = length * element size`.

Using algebra, we can rearrange this equation: `length = array size / element size`. We typically use `sizeof(array[0])` for the element size. Therefore, length = `sizeof(array) / sizeof(array[0])`. You may also occasionally see this written as `sizeof(array) / sizeof(*array)`, which does the same thing.

However, as we will show you in the next lesson, this formula can fail quite easily (when passed a decayed array), leaving the program unexpectedly broken. C++17's `std::size()` and the `length()` function template shown above will both cause compilation errors in this case, so they are safe to use.

Related content

We cover array decay in the next lesson 17.8 -- C-style array decay.

Quiz time

Question #1

Convert the following `std::array` definition to an equivalent constexpr C-style array definition:

```
constexpr std::array<int, 3> a{}; // allocate 3 ints
```

Show Solution

Question #2

What three things are wrong with the following program?

```
#include <iostream>

int main()
{
    int length{ 5 };
    const int arr[length] { 9, 7, 5, 3, 1 };

    std::cout << arr[length];
    arr[0] = 4;

    return 0;
}
```

Show Solution

Question #3

A "perfect square" is a natural number whose square root is an integer. We can make perfect squares by multiplying a natural number (including zero) by itself. The first 4 perfect squares are: 0, 1, 4, 9.

Use a global constexpr C-style array to hold the perfect squares between 0 and 9 (inclusive). Repeatedly ask the user to enter a single digit integer, or -1 to quit. Print whether the digit the user entered is a perfect square.

The output should match the following:

```
Enter a single digit integer, or -1 to quit: 4
4 is a perfect square

Enter a single digit integer, or -1 to quit: 5
5 is not a perfect square

Enter a single digit integer, or -1 to quit: -1
Bye
```

Hints: Use a range-based for loop to traverse the C-style array to look for a match.

Show Solution