17.8 — C-style array decay

The C-style array passing challenge

The designers of the C language had a problem. Consider the following simple program:

```
#include <iostream>

void print(int val)
{
    std::cout << val;
}

int main()
{
    int x { 5 };
    print(x);

    return 0;
}
```

When `print(x)` is called, the value of argument `x` (`5`) is copied into parameter `val`. Within the body of the function, the value of `val` (`5`) is printed to the console. Because `x` is cheap to copy, there's no problem here.

Now consider the following similar program, which uses a 1000 element C-style int array instead of a single int:

```
#include <iostream>

void printElementZero(int arr[1000])
{
    std::cout << arr[0]; // print the value of the first element
}

int main()
{
    int x[1000] { 5 };   // define an array with 1000 elements, x[0] is initialized
to 5
    printElementZero(x);

    return 0;
}
```

This program also compiles and prints the expected value (`5`) to the console.

While the code in this example is similar to the code in the prior example, mechanically it works a bit different than you might expect (we'll explain this below). And that is due to the solution that the C designers came up to solve for two major challenges.

First, copying a 1000 element array every time a function is called is expensive (and even more so if the elements are an expensive-to-copy type), so we want to avoid that. But how? C doesn't have references, so using pass by reference to avoid making a copy of function arguments wasn't an option.

Second, we want to be able to write a single function that can accept array arguments of different lengths. Ideally, our `printElementZero()` function in the example above should be callable with arrays arguments of any length (since element 0 is guaranteed to exist). We don't want to have to write a different function for every possible array length that we want to use as an argument. But how? C has no syntax to specify "any length" arrays, nor does it support templates, nor can arrays of one length be converted to arrays of another length (presumably because doing so would involve making an expensive copy).

The designers of the C language came up with a clever solution (inherited by C++ for compatibility reasons) that solves for both of these issues:

```
#include <iostream>

void printElementZero(int arr[1000]) // doesn't make a copy
{
    std::cout << arr[0]; // print the value of the first element
}

int main()
{
    int x[7] { 5 };      // define an array with 7 elements
    printElementZero(x); // somehow works!

    return 0;
}
```

Somehow, the above example passes a 7 element array to a function expecting a 1000 element array, without any copies being made. In this lesson, we'll explore how this works.

We'll also take a look at why the solution the C designers picked is dangerous, and not well suited for use in modern C++.

But first, we need two cover two subtopics.

Array to pointer conversions (array decay)

In most cases, when a C-style array is used in an expression, the array will be implicitly converted into a pointer to the element type, initialized with the address of the first element (with index 0). Colloquially, this is called **array decay** (or just **decay** for short).

You can see this in the following program:

```
#include <iomanip> // for std::boolalpha
#include <iostream>

int main()
{
    int arr[5]{ 9, 7, 5, 3, 1 }; // our array has elements of type int

    // First, let's prove that arr decays into an int* pointer

    auto ptr{ arr }; // evaluation causes arr to decay, type deduction should deduce
type int*
    std::cout << std::boolalpha << (typeid(ptr) == typeid(int*)) << '\n'; // Prints
true if the type of ptr is int*

    // Now let's prove that the pointer holds the address of the first element of the
array

    std::cout << std::boolalpha << (&arr[0] == ptr) << '\n';

    return 0;
}
```

On the author's machine, this printed:

```
true
true
```

There is nothing special about the pointer that an array decays into. It is a normal pointer that holds the address of the first element.

Similarly, a const array (e.g. `const int arr[5]`) decays into a pointer-to-const (`const int*`).

Tip

In C++, there are a few common cases where an C-style array doesn't decay:

1. When used as an argument to `sizeof()` or `typeid()`.
2. When taking the address of the array using `operator&`.
3. When passed as a member of a class type.
4. When passed by reference.

Because C-style arrays decay into a pointer in most cases, it's a common fallacy to believe arrays *are* pointers. This is not the case. An array object is a sequence of elements, whereas a pointer object just holds an address.

The type information of an array and a decayed array is different. In the example above, the array `arr` has type `int[5]`, whereas the decayed array has type `int*`. Notably, the array type `int[5]` contains length information, whereas the decayed array pointer type `int*` does not.

Key insight

A decayed array pointer does not know how long the array it is pointing to is. The term "decay" indicates this loss of length type information.

Subscripting a C-style array actually applies `operator[]` to the decayed pointer

Because a C-style arrays decays to a pointer when evaluated, when a C-style array is subscripted, the subscript is actually operating on the decayed array pointer:

```
#include <iostream>

int main()
{
    const int arr[] { 9, 7, 5, 3, 1 };
    std::cout << arr[2]; // subscript decayed array to get element 2, prints 5

    return 0;
}
```

We can also use `operator[]` directly on a pointer. If that pointer is holding the address of the first element, the result will be identical:

```
#include <iostream>

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

    const int* ptr{ arr };  // arr decays into a pointer
    std::cout << ptr[2];    // subscript ptr to get element 2, prints 5

    return 0;
}
```

We'll see where this is convenient in a moment, and take a deeper look at how this actually works (as well as what happens when the pointer is holding something other than the address of the first element) in the next lesson 17.9 -- Pointer arithmetic and subscripting.

Array decay solves our C-style array passing issue

Array decay solves both challenges we encountered at the top of the lesson.

When passing a C-style array as an argument, the array decays into a pointer, and the pointer holding the address of the first element of the array is what gets passed to the function. So although it looks like we're passing the C-style array by value, we're actually passing it by address! This is how making a copy of the C-style array argument is avoided.

Key insight

C-style arrays are passed by address, even when it looks like they are passed by value.

Now consider two different arrays of the same element type but different lengths (e.g. `int[5]` and `int[7]`). These are distinct types, incompatible with each other. However, they will both decay into the same pointer type (e.g. `int*`). Their decayed versions are interchangable! Dropping the length information from the type allows us to pass arrays of different lengths without a type mismatch.

Key insight

Two C-style arrays with the same element type but different lengths will decay into the same pointer type.

In the following example, we'll illustrate two things:

- That we can pass arrays of different lengths to a single function (because both decay to the same pointer type).
- That our function parameter receiving the array can be a (const) pointer of the array's element type.

```
#include <iostream>

void printElementZero(const int* arr) // pass by const address
{
    std::cout << arr[0];
}

int main()
{
    const int prime[] { 2, 3, 5, 7, 11 };
    const int squares[] { 1, 4, 9, 25, 36, 49, 64, 81 };

    printElementZero(prime);   // prime decays to an const int* pointer
    printElementZero(squares); // squares decays to an const int* pointer

    return 0;
}
```

This example works just fine, and prints:

```
2
1
```

Within `main()`, when we call `printElementZero(prime)`, the `prime` array decays from an array of type `const int[5]` to a pointer of type `const int*` that is holding the address of the first element of `prime`. Similarly, when we call `printElementZero(squares)`, `squares` decays from an array of type `const int[8]` to a pointer of type `const int*` that is holding the address of the first element of `squares`. These pointers of type `const int*` are what are actually passed to the function as an argument.

Since we're passing pointers of type `const int*`, our `printElementZero()` function needs to have a parameter of the same pointer type (`const int*`).

Within this function, we're subscripting the pointer to access the selected array element.

Because a C-style array is passed by address, the function has direct access to the array passed in (not a copy) and can modify its elements. For this reason, it's a good idea to make sure your function parameter is const if your function does not intend to modify the array elements.

C-style array function parameter syntax

One problem with declaring the function parameter as `int* arr` is that it's not obvious that `arr` is supposed to be a pointer to an array of values rather than a pointer to a single integer. For this reason, when passing a C-style array, its preferable to use the alternate declaration form `int arr[]`:

```
#include <iostream>

void printElementZero(const int arr[]) // treated the same as const int*
{
    std::cout << arr[0];
}

int main()
{
    const int prime[] { 2, 3, 5, 7, 11 };
    const int squares[] { 1, 4, 9, 25, 36, 49, 64, 81 };

    printElementZero(prime);  // prime decays to a pointer
    printElementZero(squares); // squares decays to a pointer

    return 0;
}
```

This program behaves identically to the prior one, as the compiler will interpret function parameter `const int arr[]` the same as `const int*`. However, this has the advantage of communicating to the caller that `arr` is expected to be a decayed C-style array, not a pointer to a single value. Note that no length information is required between the square brackets (since it is not used anyway). If a length is provided, it will be ignored.

Best practice

A function parameter expecting a C-style array type should use the array syntax (e.g. `int arr[]`) rather than pointer syntax (e.g. `int *arr`).

The downside of using this syntax is that it makes it less obvious that `arr` has decayed (whereas it's quite clear with the pointer syntax), so you'll need to take extra care not to do anything that doesn't work as expected with a decayed array (we'll cover some of these in a moment).

The problems with array decay

Although array decay was a clever solution to ensure C-style arrays of different lengths could be passed to a function without making expensive copies, the loss of array length information makes it easy for several types of mistakes to be made.

First, `sizeof()` will return different values for arrays and decayed arrays:

```
#include <iostream>

void printArraySize(int arr[])
{
    std::cout << sizeof(arr) << '\n'; // prints 4 (assuming 32-bit addresses)
}

int main()
{
    int arr[]{ 3, 2, 1 };

    std::cout << sizeof(arr) << '\n'; // prints 12 (assuming 4 byte ints)

    printArraySize(arr);

    return 0;
}
```

This means using `sizeof()` on a C-style array is potentially dangerous, as you have to ensure you are using it only when you can access the actual array object, not the decayed array or pointer.

In the prior lesson (17.7 -- Introduction to C-style arrays), we mentioned that `sizeof(arr)/sizeof(*arr)` was historically used as a hack to get the size of a C-style array. This hack is dangerous because if `arr` has decayed, `sizeof(arr)` will return the size of a pointer rather than the size of the array, producing the wrong array length as a result, likely causing the program to malfunction.

Fortunately, C++17's better replacement `std::size()` (and C++20's `std::ssize()`) will fail to compile if passed a pointer value:

```
#include <iostream>

int printArrayLength(int arr[])
{
    std::cout << std::size(arr) << '\n'; // compile error: std::size() won't work on a pointer
}

int main()
{
    int arr[]{ 3, 2, 1 };

    std::cout << std::size(arr) << '\n'; // prints 3

    printArrayLength(arr);

    return 0;
}
```

Second, and perhaps most importantly, array decay can make refactoring (breaking long functions into shorter, more modular functions) difficult. Code that works as expected with a non-decayed array may not compile (or worse, may silently malfunction) when the same code is using a decayed array.

Third, not having length information creates several programmatic challenges. Without length information, it is different to sanity check the length of the array. Users can easily pass in arrays that are shorter than expected (or even pointers to a single value), which will then cause undefined behavior when they are subscripted with an invalid index.

```
#include <iostream>

void printElement2(int arr[])
{
    // How do we ensure that arr has at least three elements?
    std::cout << arr[2] << '\n';
}

int main()
{
    int a[]{ 3, 2, 1 };
    printElement2(a);  // ok

    int b[]{ 7, 6 };
    printElement2(b);  // compiles but produces undefined behavior

    int c{ 9 };
    printElement2(&c); // compiles but produces undefined behavior

    return 0;
}
```

Not having the array length also create challenges when traversing the array -- how do we know when we've reached the end?

There are solutions to these issues, but these solutions add both complexity and fragility to a program.

Working around array length issues

Historically, programmers have worked around the lack of array length information via one of two methods.

First, we can pass in both the array and the array length as separate arguments:

```
#include <cassert>
#include <iostream>

void printElement2(const int arr[], int length)
{
    assert(length > 2 && "printElement2: Array too short"); // can't static_assert on
length

    std::cout << arr[2] << '\n';
}

int main()
{
    constexpr int a[]{ 3, 2, 1 };
    printElement2(a, static_cast<int>(std::size(a)));  // ok

    constexpr int b[]{ 7, 6 };
    printElement2(b, static_cast<int>(std::size(b)));  // will trigger assert

    return 0;
}
```

However, this still has a number of issues:

- The caller needs to make sure that the array and the array length are paired -- if the wrong value for length is passed in, the function will still malfunction.
- There are potential sign conversion issues if you're using `std::size()` or a function that returns a length as `std::size_t`.
- Runtime asserts only trigger when encountered at runtime. If our testing path doesn't cover all calls to the function, there's a risk of shipping a program to the customer that will assert when they do something we didn't explicitly test for. In modern C++, we'd want to use `static_assert` for compile-time validation of the array length of constexpr arrays, but there's no easy way to do this (as function parameters can't be constexpr, even in constexpr or consteval functions!).
- This method only works if we're making an explicit function call. If the function call is implicit (e.g. we're calling an operator with the array as an operand), then there's no opportunity to pass in the length.

Second, if there is an element value that is not semantically valid (e.g. a test score of `-1`), we can instead mark the end of the array using an element of that value. That way, the length of the array can be calculated by counting how many elements exist between the start of the array and this terminating element. The array can also be traversed by iterating from the start until we hit the terminating element. The nice thing about this method is that it works even with implicit function calls.

Key insight

C-style strings (which are C-style arrays) use a null-terminator to mark the end of the string, so that they can be traversed even if they have decayed.

But this method also has a number of issues:

- If the terminating element does not exist, traversal will walk right off the end of the array, causing undefined behavior.
- Functions that traverse the array need special handling for the terminating element (e.g. a C-style string print function needs to know not to print the terminating element).
- There is a mismatch between the actual array length and the number of semantically valid elements. If the wrong length is used, the semantically invalid terminating element may be "processed".
- This approach only works if a semantically invalid value exists, which is often not the case.

C-style arrays should be avoided in most cases

Because of the non-standard passing semantics (pass by address is used instead of pass by value) and risks associated with decayed arrays losing their length information, C-style arrays have generally fallen out of favor. We recommend avoiding them as much as possible.

Best practice

Avoid C-style arrays whenever practical.

- Prefer `std::string_view` for read-only strings (string literal symbolic constants and string parameters).
- Prefer `std::string` for modifiable strings.
- Prefer `std::array` for non-global constexpr arrays.
- Prefer `std::vector` for non-constexpr arrays.

It is okay to use C-style arrays for global constexpr arrays. We'll discuss this in a moment.

As an aside…

In C++, arrays can be passed by reference, in which case the array argument won't decay when passed to a function (but the reference to the array will still decay when evaluated). However, it's easy to forget to apply this consistently, and one missed reference will lead to decaying arguments. Also, array reference parameters must have a fixed length, meaning the function can only handle arrays of one particular length. If we want a function that can handle arrays of different lengths, then we also have to use function templates. But if you're going to do both of those things to "fix" C-style arrays, then you might as well just use `std::array`!

So when are C-style arrays used in modern C++?

In modern C++, C-style arrays are typically used in two cases:

1. To store constexpr global (or constexpr static local) program data. Because such arrays can be accessed directly from anywhere in the program, we do not need to pass the array, which avoids decay-related issues. The syntax for defining C-style arrays can be a little less wonky than `std::array`. More importantly, indexing such arrays does not have sign conversion issues like the standard library container classes do.
2. As parameters to functions or classes that want to handle non-constexpr C-style string arguments directly (rather than requiring a conversion to `std::string_view`). There are two possible reasons for this: First, converting a non-constexpr C-style string to a `std::string_view` requires traversing the C-style string to determine its length. If the function is in a performance critical section of code and the length isn't needed (e.g. because the function is going to traverse the string anyway) then avoiding the conversion may be useful. Second, if the function (or class) calls other functions that expect C-style strings, converting to a `std::string_view` just to convert back may be suboptimal (unless you have other reasons for wanting a `std::string_view`).

Quiz time

Question #1

What is array decay, and why is it a problem?

Show Solution

Question #2

Why do C-style strings (which are C-style arrays) use a null terminator?

Show Solution

Question #3

Extra credit: Why do C-style strings use a null-terminator instead of requiring both the decayed C-style string and explicit length information to be passed to a function?

Show Solution

Extra Credit #2: Even if C++ wanted to implement having to pass explicit length information, why wouldn't it work?

Show Hint

Show Solution