17.3 — Passing and returning std::array

An object of type `std::array` can be passed to a function just like any other object. That means if we pass a `std::array` by value, an expensive copy will be made. Therefore, we typically pass `std::array` by (const) reference to avoid such copies.

With a `std::array`, both the element type and array length are part of the type information of the object. Therefore, when we use a `std::array` as a function parameter, we have to explicitly specify both the element type and array length:

```
#include <array>
#include <iostream>

void passByRef(const std::array<int, 5>& arr) // we must explicitly specify <int, 5>
here
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 }; // CTAD deduces type std::array<int, 5>
    passByRef(arr);

    return 0;
}
```

CTAD doesn't (currently) work with function parameters, so we cannot just specify `std::array` here and let the compiler deduce the template arguments.

Using function templates to pass `std::array` of different element types or lengths

To write a function that can accept `std::array` with any kind of element type or any length, we can create a function template that parameterizes both the element type and length of our `std::array`, and then C++ will use that function template to instantiate real functions with actual types and lengths.

Related content

We cover function templates in lesson 11.6 -- Function templates.

Since `std::array` is defined like this:

```
template<typename T, std::size_t N> // N is a non-type template parameter
struct array;
```

We can create a function template that uses the same template parameter declaration:

```
#include <array>
#include <iostream>

template <typename T, std::size_t N> // note that this template parameter declaration
matches the one for std::array
void passByRef(const std::array<T, N>& arr)
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 }; // use CTAD to infer std::array<int, 5>
    passByRef(arr);  // ok: compiler will instantiate passByRef(const std::array<int,
5>& arr)

    std::array arr2{ 1, 2, 3, 4, 5, 6 }; // use CTAD to infer std::array<int, 6>
    passByRef(arr2); // ok: compiler will instantiate passByRef(const std::array<int,
6>& arr)

    std::array arr3{ 1.2, 3.4, 5.6, 7.8, 9.9 }; // use CTAD to infer
std::array<double, 5>
    passByRef(arr3); // ok: compiler will instantiate passByRef(const
std::array<double, 5>& arr)

    return 0;
}
```

In the above example, we've created a single function template named `passByRef()` that has a parameter of type `std::array<T, N>`. `T` and `N` are defined in the template parameter declaration on the previous line: `template <typename T, std::size_t N>`. `T` is a standard type template parameter that allows the caller to specify the element type. `N` is a non-type template parameter of type `std::size_t` that allows the caller to specify the array length.

Warning

Note that the type of the non-type template parameter for `std::array` should be `std::size_t`, not `int`! This is because `std::array` is defined as `template<class T, std::size_t N> struct array;`. If you use `int` as the type of the non-type template parameter, the compiler will be unable to match the argument of type `std::array<T, std::size_t>` with the parameter of type `std::array<T, int>` (and templates won't do conversions).

Therefore, when we call `passByRef(arr)` from `main()` (where `arr` is defined as a `std::array<int, 5>`), the compiler will instantiate and call `void passByRef(const std::array<int, 5>& arr)`. A similar process happens for `arr2` and `arr3`.

Thus, we’ve created a single function template that can instantiate functions to handle `std::array` arguments of any element type and length!

If desired, it is also possible to only template one of the two template parameters. In the following example, we parameterize only the length of the `std::array`, but the element type is explicitly defined as `int`:

```
#include <array>
#include <iostream>

template <std::size_t N> // note: only the length has been templated here
void passByRef(const std::array<int, N>& arr) // we've defined the element type as
int
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 }; // use CTAD to infer std::array<int, 5>
    passByRef(arr);  // ok: compiler will instantiate passByRef(const std::array<int,
5>& arr)

    std::array arr2{ 1, 2, 3, 4, 5, 6 }; // use CTAD to infer std::array<int, 6>
    passByRef(arr2); // ok: compiler will instantiate passByRef(const std::array<int,
6>& arr)

    std::array arr3{ 1.2, 3.4, 5.6, 7.8, 9.9 }; // use CTAD to infer
std::array<double, 5>
    passByRef(arr3); // error: compiler can't find matching function

    return 0;
}
```

Auto non-type template parameters C++20

Having to remember (or look up) the type of a non-type template parameter so that you can use it in a template parameter declaration for your own function templates is a pain.

In C++20, we can use `auto` in a template parameter declaration to have a non-type template parameter deduce its type from the argument:

```
#include <array>
#include <iostream>

template <typename T, auto N> // now using auto to deduce type of N
void passByRef(const std::array<T, N>& arr)
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 }; // use CTAD to infer std::array<int, 5>
    passByRef(arr);  // ok: compiler will instantiate passByRef(const std::array<int,
5>& arr)

    std::array arr2{ 1, 2, 3, 4, 5, 6 }; // use CTAD to infer std::array<int, 6>
    passByRef(arr2); // ok: compiler will instantiate passByRef(const std::array<int,
6>& arr)

    std::array arr3{ 1.2, 3.4, 5.6, 7.8, 9.9 }; // use CTAD to infer
std::array<double, 5>
    passByRef(arr3); // ok: compiler will instantiate passByRef(const
std::array<double, 5>& arr)

    return 0;
}
```

If your compiler is C++20 capable, this is fine to use.

Static asserting on array length

Consider the following template function, which is similar to the one presented above:

```
#include <array>
#include <iostream>

template <typename T, std::size_t N>
void printElement3(const std::array<T, N>& arr)
{
    std::cout << arr[3] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 };
    printElement3(arr);

    return 0;
}
```

While `printElement3()` works fine in this case, there's a potential bug waiting for a unwary programmer in this program. See it?

The above program prints the value of the array element with index 3. This is fine as long as the array has a valid element with index 3. However, the compiler will happily let you pass in arrays where index 3 is out of bounds. For example:

```
#include <array>
#include <iostream>

template <typename T, std::size_t N>
void printElement3(const std::array<T, N>& arr)
{
    std::cout << arr[3] << '\n'; // invalid index
}

int main()
{
    std::array arr{ 9, 7 }; // a 2-element array (valid indexes 0 and 1)
    printElement3(arr);

    return 0;
}
```

This leads to undefined behavior. Ideally, we'd like the compiler to warn us when we try to do something like this!

One advantage that template parameters have over function parameters is that template parameters are compile-time constants. This means we can take advantage of capabilities that require constant expressions.

So one solution is to use `std::get()` (which does compile-time bounds checking) instead of `operator[]` (which does no bounds checking):

```
#include <array>
#include <iostream>

template <typename T, std::size_t N>
void printElement3(const std::array<T, N>& arr)
{
    std::cout << std::get<3>(arr) << '\n'; // checks that index 3 is valid at
compile-time
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 };
    printElement3(arr); // okay

    std::array arr2{ 9, 7 };
    printElement3(arr2); // compile error

    return 0;
}
```

When the compiler reaches the call to `printElement3(arr2)`, it will instantiate the function `printElement3(const std::array<int, 2>&)`. Inside the body of this function is the line `std::get<3>(arr)`. Since the array parameter's length is 2, this is a invalid access, and the compiler will emit an error.

An alternative solution is to use `static_assert` to validate a precondition on the array length ourselves:

Related content

We cover preconditions in lesson 9.6 -- Assert and static_assert.

```
#include <array>
#include <iostream>

template <typename T, std::size_t N>
void printElement3(const std::array<T, N>& arr)
{
    // precondition: array length must be greater than 3 so element 3 exists
    static_assert (N > 3);

    // we can assume the array length is greater than 3 beyond this point

    std::cout << arr[3] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 };
    printElement3(arr); // okay

    std::array arr2{ 9, 7 };
    printElement3(arr2); // compile error

    return 0;
}
```

When the compiler reaches the call to `printElement3(arr2)`, it will instantiate the function `printElement3(const std::array<int, 2>&)`. Inside the body of this function is the line `static_assert (N > 3)`. Since the `N` template non-type parameter has value `2`, and `2 > 3` is false, the compiler will emit an error.

Key insight

In the example above, you may be wondering why we use `static_assert (N > 3);` instead of `static_assert (std::size(arr) > 3)`. The latter won't compile prior to C++23 due to the language defect mentioned in the prior lesson (17.2 -- std::array length and indexing).

Returning a `std::array`

Syntax aside, passing a `std::array` to a function is conceptually simple -- pass it by (const) reference. But what if we have a function that needs to return a `std::array`? Things are a little more complicated. Unlike `std::vector`, `std::array` is not move-capable, so returning a `std::array` by value will make a copy of the array. The elements inside the array will be moved if they are move-capable, and copied otherwise.

There are two conventional options here, and which you should pick depends on circumstances.

Returning a `std::array` by value

It is okay to return a `std:array` by value when all of the following are true:

- The array isn't huge.
- The element type is cheap to copy (or move).
- The code isn't being used in a performance-sensitive context.

In such cases, a copy of the `std::array` will be made, but if all of the above are true, the performance hit will be minor, and sticking with the most conventional way to return data to the caller may be the best choice.

```
#include <array>
#include <iostream>
#include <limits>

// return by value
template <typename T, std::size_t N>
std::array<T, N> inputArray() // return by value
{
	std::array<T, N> arr{};
	std::size_t index { 0 };
	while (index < N)
	{
		std::cout << "Enter value #" << index << ": ";
		std::cin >> arr[index];

		if (!std::cin) // handle bad input
		{
			std::cin.clear();
			std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
			continue;
		}
		++index;
	}

	return arr;
}

int main()
{
	std::array<int, 5> arr { inputArray<int, 5>() };

	std::cout << "The value of element 2 is " << arr[2] << '\n';

	return 0;
}
```

There are a few nice things about this method:

- It uses the most conventional way to return data to the caller.

- It’s obvious that the function is returning a value.
- We can define an array and use the function to initialize it in a single statement.

There are also a few downsides:

- The function returns a copy of the array and all its elements, which isn’t cheap.
- When we call the function, we must explicitly supply the template arguments since there is no parameter to deduce them from.

Returning a `std::array` via an out parameter

In cases where return by value is too expensive, we can use an out-parameter instead. In this case, the caller is responsible for passing in the `std::array` by non-const reference (or by address), and the function can then modify this array.

```
#include <array>
#include <limits>
#include <iostream>

template <typename T, std::size_t N>
void inputArray(std::array<T, N>& arr) // pass by non-const reference
{
	std::size_t index { 0 };
	while (index < N)
	{
		std::cout << "Enter value #" << index << ": ";
		std::cin >> arr[index];

		if (!std::cin) // handle bad input
		{
			std::cin.clear();
			std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
			continue;
		}
		++index;
	}

}

int main()
{
	std::array<int, 5> arr {};
	inputArray(arr);

	std::cout << "The value of element 2 is " << arr[2] << '\n';

	return 0;
}
```

The primary advantage of this method is that no copy of the array is ever made, so this is efficient.

There are also a few downsides:

- This method of returning data is non-conventional, and it is not easy to tell that the function is modifying the argument.
- We can only use this method to assign values to the array, not initialize it.
- Such a function cannot be used to produce temporary objects.

Returning a `std::vector` instead

`std::vector` is move-capable and can be returned by value without making expensive copies. If you're returning a `std::array` by value, your `std::array` probably isn't constexpr, and you should consider using (and returning) `std::vector` instead.

Quiz time

Question #1

Complete the following program:

```
#include <array>
#include <iostream>

int main()
{
    constexpr std::array arr1 { 1, 4, 9, 16 };
    printArray(arr1);

    constexpr std::array arr2 { 'h', 'e', 'l', 'l', 'o' };
    printArray(arr2);

    return 0;
}
```

When run, it should print:

```
The array (1, 4, 9, 16) has length 4
The array (h, e, l, l, o) has length 5
```

Show Solution