17.4 — std::array of class types, and brace elision

A `std::array` isn't limited to elements of fundamental types. Rather, the elements of a `std::array` can be any object type, including compound types. This means you can create a `std::array` of pointers, or a `std::array` of structs (or classes)

However, initializing a `std::array` of structs or classes tends to trip new programmers up, so we're going to spend a lesson explicitly covering this topic.

Author's note

We'll use structs to illustrate our points in this lesson. The material applies equally well to classes.

Defining and assigning to a `std::array` of structs

Let's start with a simple struct:

```
struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};
```

Defining a `std::array` of `House` and assigning elements works just like you'd expect:

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    std::array<House, 3> houses{};

    houses[0] = { 13, 1, 7 };
    houses[1] = { 14, 2, 5 };
    houses[2] = { 15, 2, 4 };

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories * house.roomsPerStory)
                  << " rooms.\n";
    }

    return 0;
}
```

The above outputs the following:

```
House number 13 has 7 rooms.
House number 14 has 10 rooms.
House number 15 has 8 rooms.
```

Initializing a `std::array` of structs

Initializing an array of structs also works just like you'd expect, so long as you are explicit about the element type:

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    constexpr std::array houses { // use CTAD to deduce template arguments <House, 3>
            House{ 13, 1, 7 },
            House{ 14, 2, 5 },
            House{ 15, 2, 4 }
        };

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
            << " has " << (house.stories * house.roomsPerStory)
            << " rooms.\n";
    }

    return 0;
}
```

In the above example, we're using CTAD to deduce the type of the `std::array` as `std::array<House, 3>`. We then provide 3 `House` objects as initializers, which works just fine.

Initialization without explicitly specifying the element type for each initializer

In the above example, you'll note that each initializer requires us to list the element type:

```
constexpr std::array houses {
        House{ 13, 1, 7 }, // we mention House here
        House{ 14, 2, 5 }, // and here
        House{ 15, 2, 4 }  // and here
    };
```

But we did not have to do the same in the assignment case:

```
// The compiler knows that each element of houses is a House
// so it will implicitly convert the right hand side of each assignment to a House
houses[0] = { 13, 1, 7 };
houses[1] = { 14, 2, 5 };
houses[2] = { 15, 2, 4 };
```

So you might think to try something like this:

```
// doesn't work
constexpr std::array<House, 3> houses { // we're telling the compiler that each
element is a House
        { 13, 1, 7 }, // but not mentioning it here
        { 14, 2, 5 },
        { 15, 2, 4 }
    };
```

Perhaps surprisingly, this doesn't work. Let's explore why.

A `std::array` is defined as a struct that contains a single C-style array member (whose name is implementation defined), like this:

```
template<typename T, std::size_t N>
struct array
{
    T implementation_defined_name[N]; // a C-style array with N elements of type T
}
```

Author's note

We haven't covered C-style arrays yet, but for the purposes of this lesson, all you need to know is that `T implementation_defined_name[N];` is a fixed-size array of N elements of type T (just like `std::array<T, N> implementation_defined_name;`).

We cover C-style arrays in upcoming lesson 17.7 -- Introduction to C-style arrays.

So when we try to initialize `houses` per the above, the compiler interprets the initialization like this:

```
// Doesn't work
constexpr std::array<House, 3> houses { // initializer for houses
    { 13, 1, 7 }, // initializer for C-style array member with
implementation_defined_name
    { 14, 2, 5 }, // ?
    { 15, 2, 4 }  // ?
};
```

The compiler will interpret `{ 13, 1, 7 }` as the initializer for the first member of `houses`, which is the C-style array with the implementation defined name. This will initialize the C-style array element 0 with `{ 13, 1, 7 }` and the rest of the members will be zero-initialized. Then the compiler will discover we've provided two more initialization values (`{ 14, 2, 7 }` and `{ 15, 2, 5 }`) and produce a compilation error telling us that we've provided too many initialization values.

The correct way to initialize the above is to add an extra set of braces as follows:

```
// This works as expected
constexpr std::array<House, 3> houses { // initializer for houses
    { // extra set of braces to initialize the C-style array member with
implementation_defined_name
        { 13, 4, 30 }, // initializer for array element 0
        { 14, 3, 10 }, // initializer for array element 1
        { 15, 3, 40 }, // initializer for array element 2
     }
};
```

Note the extra set of braces that are required (to begin initialization of the C-style array member inside the `std::array` struct). Within those braces, we can then initialize each element individually, each inside its own set of braces.

This is why you'll see `std::array` initializers with an extra set of braces when the element type requires a list of values and we are not explicitly providing the element type as part of the initializer.

Key insight

When initializing a `std::array` with a struct, class, or array and not providing the element type with each initializer, you'll need an extra pair of braces so that the compiler will properly interpret what to initialize.

This is an artifact of aggregate initialization, and other standard library container types (that use list constructors) do not require the double braces in these cases.

Here's a full example:

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    constexpr std::array<House, 3> houses {{ // note double braces
        { 13, 1, 7 },
        { 14, 2, 5 },
        { 15, 2, 4 }
    }};

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories * house.roomsPerStory)
                  << " rooms.\n";
    }

    return 0;
}
```

Brace elision for aggregates

Given the explanation above, you may be wondering why the above case requires double braces, but all other cases we've seen only require single braces:

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

int main()
{
    constexpr std::array<int, 5> arr { 1, 2, 3, 4, 5 }; // single braces

    for (const auto n : arr)
        std::cout << n << '\n';

    return 0;
}
```

It turns out that you can supply double braces for such arrays:

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

int main()
{
    constexpr std::array<int, 5> arr {{ 1, 2, 3, 4, 5 }}; // double braces

    for (const auto n : arr)
        std::cout << n << '\n';

    return 0;
}
```

However, aggregates in C++ support a concept called **brace elision**, which lays out some rules for when multiple braces may be omitted. Generally, you can omit braces when initializing a `std::array` with scalar (single) values, or when initializing with class types or arrays where the type is explicitly named with each element.

There is no harm in always initializing `std::array` with double braces, as it avoids having to think about whether brace-elision is applicable in a specific case or not. Alternatively, you can try to single-brace init, and the compiler will generally complain if it can't figure it out. In that case, you can quickly add an extra set of braces.

Another example

Here's one more example where we initialize a `std::array` with `Student` structs.

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

// Each student has an id and a name
struct Student
{
	int id{};
	std::string_view name{};
};

// Our array of 3 students (single braced since we mention Student with each
initializer)
constexpr std::array students{ Student{0, "Alex"}, Student{ 1, "Joe" }, Student{ 2,
"Bob" } };

const Student* findStudentById(int id)
{
	// Look through all the students
	for (auto& s : students)
	{
		// Return student with matching id
		if (s.id == id) return &s;
	}

	// No matching id found
	return nullptr;
}

int main()
{
	constexpr std::string_view nobody { "nobody" };

	const Student* s1 { findStudentById(1) };
	std::cout << "You found: " << (s1 ? s1->name : nobody) << '\n';

	const Student* s2 { findStudentById(3) };
	std::cout << "You found: " << (s2 ? s2->name : nobody) << '\n';

	return 0;
}
```

This prints:

```
You found: Joe
You found: nobody
```

Note that because `std::array students` is constexpr, our `findStudentById()` function must return a const pointer, which means our `Student` pointers in `main()` must also be const.

Quiz time

Question #1

Define a struct named `Item` that contains two members: `std::string_view name` and `int gold`. Define a `std::array` and initialize it with 4 Item objects, explicitly specifying the element type for each initializer.

The program should print the following:

```
A sword costs 5 gold.
A dagger costs 3 gold.
A club costs 2 gold.
A spear costs 7 gold.
```

Show Solution

Question #2

Update your solution to quiz 1 to not explicitly specify the element type for each initializer.

Show Solution