LEARN C++

Skill up with our free tutorials

17.5 — Arrays of references via std::reference_wrapper

ALEX **OCTOBER 14, 2023**

In the prior lesson, we mentioned that arrays can have elements of any object type. This includes objects with fundamental types (e.g. int) and objects with compound types (e.g. pointer to int).

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

int main()
{
    int x { 1 };
    int y { 2 };

    [[maybe_unused]] std::array valarr { x, y };   // an array of int values
    [[maybe_unused]] std::vector ptrarr { &x, &y }; // a vector of int pointers

    return 0;
}
```

However, because references are not objects, you cannot make an array of references. The elements of an array must also be assignable, and references can't be reseated.

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

int main()
{
    int x { 1 };
    int y { 2 };

    [[maybe_unused]] std::array<int&, 2> refarr { x, y }; // compile error: cannot define array of references

    int& ref1 { x };
    int& ref2 { y };
    [[maybe_unused]] std::array valarr { ref1, ref2 }; // ok: this is actually a std::array<int, 2>, not an array of
references

    return 0;
}
```

In this lesson, we'll use `std::array` in the examples, but this is equally applicable to all array types.

However, if you want an array of references, there is a workaround.

std::reference_wrapper

`std::reference_wrapper` is a standard library class template that lives in the <functional> header. It takes a type template argument T, and then behaves like a modifiable lvalue reference to T.

There are a few things worth noting about `std::reference_wrapper`:

- `Operator=` will reseat a `std::reference_wrapper` (change which object is being referenced).
- `std::reference_wrapper<T>` will implicitly convert to `T&`.
- The `get()` member function can be used to get a `T&`. This is useful when we want to update the value of the object being referenced.

Here's a simple example:

```
#include <array>
#include <functional> // for std::reference_wrapper
#include <iostream>

int main()
{
    int x { 1 };
    int y { 2 };
    int z { 3 };

    std::array<std::reference_wrapper<int>, 3> arr { x, y, z };

    arr[1].get() = 5; // modify the object in array element 1

    std::cout << arr[1] << y << '\n'; // show that we modified arr[1] and y, prints 55

    return 0;
}
```

This example prints the following:

```
55
```

Note that we must use `arr[1].get() = 5` and not `arr[1] = 5`. The latter is ambiguous, as the compiler can't tell if we intend to reseat the `std::reference_wrapper<int>` to value 5 (something that is illegal anyway) or change the value being referenced. Using `get()` disambiguates this.

When printing `arr[1]`, the compiler will realize it can't print a `std::reference_wrapper<int>`, so it will implicitly convert it to an `int&`, which it can print. So we don't need to use `get()` here.

`std::ref` and `std::cref`

Prior to C++17, CTAD (class template argument deduction) didn't exist, so all template arguments for a class type needed to be listed explicitly. Thus, to create a `std::reference_wrapper<int>`, you could do either of these:

```
    int x { 5 };

    std::reference_wrapper<int> ref1 { x };        // C++11
    auto ref2 { std::reference_wrapper<int>{ x }}; // C++11
```

Between the long name and having to explicitly list the template arguments, creating many such reference wrappers could be a pain.

To make things easier, the `std::ref()` and `std::cref()` functions were provided as shortcuts to create `std::reference_wrapper` and `const std::reference_wrapper` wrapped objects. Note that these functions can be used with `auto` to avoid having to explicitly specify the template argument.

```
        int x { 5 };
        auto ref { std::ref(x) };   // C++11, deduces to std::reference_wrapper<int>
        auto cref { std::cref(x) }; // C++11, deduces to std::reference_wrapper<const int>
```

Of course, now that we have CTAD in C++17, we can also do this:

```
        std::reference_wrapper ref1 { x };        // C++17
        auto ref2 { std::reference_wrapper{ x }}; // C++17
```

But since `std::ref()` and `std::cref()` are shorter to type, they are still widely used to create `std::reference_wrapper` objects.

Next lesson
17.6 std::array and enumerations

Back to table of contents

Previous lesson
17.4 std::array of class types, and brace elision