16.4 — Passing std::vector

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

With a `std::vector`, the element type is part of the type information of the object. Therefore, when we use a `std::vector` as a function parameter, we have to explicitly specify the element type:

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

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

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes);

    return 0;
}
```

Passing `std::vector` of different element types

Because our `passByRef()` function expects a `std::vector<int>`, we are unable to pass vectors with different element types:

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

void passByRef(const std::vector<int>& arr)
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes);  // ok: this is a std::vector<int>

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl); // compile error: std::vector<double> is not convertible to
std::vector<int>

    return 0;
}
```

In C++17 or newer, you might try to use CTAD to solve this problem:

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

void passByRef(const std::vector& arr) // compile error: CTAD can't be used to infer
function parameters
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 }; // okay: use CTAD to infer std::vector<int>
    passByRef(primes);

    return 0;
}
```

Although CTAD will work to deduce an vector's element type from initializers when it is defined, CTAD doesn't (currently) work with function parameters.

We've seen this kind of problem before, where we have overloaded functions that only differ by the parameter type. This is a great place to make use of function templates! We can create a function template that parameterizes the element type, and then C++ will use that function template to instantiate functions with actual types.

Related content

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

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

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

template <typename T>
void passByRef(const std::vector<T>& arr)
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes); // ok: compiler will instantiate passByRef(const
std::vector<int>&)

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl);    // ok: compiler will instantiate passByRef(const
std::vector<double>&)

    return 0;
}
```

In the above example, we've created a single function template named `passByRef()` that has a parameter of type `const std::vector<T>&`. `T` is defined in the template parameter declaration on the previous line: `template <typename T`. `T` is a standard type template parameter that allows the caller to specify the element type.

Therefore, when we call `passByRef(primes)` from `main()` (where `primes` is defined as a `std::vector<int>`), the compiler will instantiate and call `void passByRef(const std::vector<int>& arr)`.

When we call `passByRef(dbl)` from `main()` (where `dbl` is defined as a `std::vector<double>`), the compiler will instantiate and call `void passByRef(const std::vector<double>& arr)`.

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

Passing a `std::vector` using a generic template or abbreviated function template

We can also create a function template that will accept any type of object:

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

template <typename T>
void passByRef(const T& arr) // will accept any type of object that has an overloaded
operator[]
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes); // ok: compiler will instantiate passByRef(const
std::vector<int>&)

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl);    // ok: compiler will instantiate passByRef(const
std::vector<double>&)

    return 0;
}
```

In C++20, we can use an abbreviated function template (via an `auto` parameter) to do the same thing:

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

void passByRef(const auto& arr) // abbreviated function template
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes); // ok: compiler will instantiate passByRef(const
std::vector<int>&)

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl);    // ok: compiler will instantiate passByRef(const
std::vector<double>&)

    return 0;
}
```

Both of these will accept an argument of *any* type that will compile. This can be desirable when writing functions that we might want to operate on more than just a `std::vector`. For example, the above functions will also work on a `std::array`, a `std::string`, or some other type we may not have even considered.

The potential downside of this method is that it may lead to bugs if the function is passed an object of a type that compiles but doesn't make sense semantically.

Asserting on array length

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

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

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

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

    return 0;
}
```

While `printElement3(arr)` 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 <iostream>
#include <vector>

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

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

    return 0;
}
```

This leads to undefined behavior.

One option here is to assert on `arr.size()`, which will catch such errors when run in a debug build configuration. Because `std::vector::size()` is a non-constexpr function, we can only do a runtime assert here.

Tip

A better option is to avoid using `std::vector`in cases where you need to assert on array length. Using a type that supports `constexpr` arrays (e.g. `std::array`) is probably a better choice, as you can `static_assert` on the length of a constexpr array. We cover this in future lesson 17.3 -- Passing and returning std::array.

The best option is to avoid writing functions that rely on the user passing in a vector with a minimum length in the first place.

Quiz time

Question #1

Write a function that takes two parameters: a `std::vector` and an index. If the index is out of bounds, print an error. If the index is in bounds, print the value of the element.

The following sample program should compile:

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

// Write your printElement function here

int main()
{
    std::vector v1 { 0, 1, 2, 3, 4 };
    printElement(v1, 2);
    printElement(v1, 5);

    std::vector v2 { 1.1, 2.2, 3.3 };
    printElement(v2, 0);
    printElement(v2, -1);

    return 0;
}
```

and produce the following result:

```
The element has value 2
Invalid index
The element has value 1.1
Invalid index
```

Show Solution