On writing functions that accept any specialization of a C++ template type

devblogs.microsoft.com/oldnewthing/20230529-00

May 29, 2023

Raymond Chen

Suppose you want to write a template function that accepts any specialization of `std::vector`. Your first try would probably be something like this:

```
template<typename Value>
void accept_any_vector(std::vector<Value> v);
```

However, this does not actually accept any vector specialization. There is a second template argument for the allocator, which has a default value that nearly everyone uses. But if somebody has a vector with a custom allocator, then your function won't match.

Okay, so add an allocator, too.

```
template<typename Value, typename Allocator>
void accept_any_vector(std::vector<Value, Allocator> v);
```

This works today, but it may not work tomorrow.

A future version of `std::vector` might add new template arguments, provided that they have suitable defaults that preserve existing behavior.[1] We don't want that future version to break us, so we should slurp up all the template arguments that exist:

```
template<typename...Args>
void accept_any_vector(std::vector<Args...> v);
```

Now, once we've accepted any kind of vector, we have lost the template parameters that name the value type and allocator. You might think you could rescue them by naming them and putting all the future nonsense in the variadic portion:

```
template<typename Value, typename Allocator, typename...Extra>
void accept_any_vector(std::vector<Value, Allocator, Extra...> v);
```

However, this doesn't work:

```
// msvc
error C2977: 'std::vector': too many template arguments

// clang
error: too many template arguments for class template 'vector'

// icc
error: too many arguments for class template "std::vector"

// gcc
(no errors)
```

Fortunately, you can recover the `Value` and `Allocator` from `std::vector<Args...>` because `std::vector` lets you access the underlying type and allocator through member types.

```
template<typename...Args>
void accept_any_vector(std::vector<Args...> v)
{
    using vector = std::vector<Args...>;
    using Value = typename vector::value_type;
    using Allocator = typename vector::allocator_type;

    ...
}
```

C++ standard library types generally provide these member types to allow you to recover the template type arguments from the type. Other libraries are hit or miss.

[1] The standard also permits functions to accept default parameters or have additional default template arguments.[2] This is used primarily for SFINAE purposes, so that some overloads become removed from consideration if particular requirements are not met. The standard has the concept of “addressable functions”, which are the functions that the standard guarantees will never be overloaded or have a signature different from the one printed in the standard.

[2] We ran afoul of this issue in an April Fool’s article about invoke-oriented programming.