How do I write a function that accepts any type of standard container?

devblogs.microsoft.com/oldnewthing/20190619-00

June 19, 2019

Raymond Chen

Suppose you want a function to accept any sort of standard container. You just want a bunch of, say, integers, and it could arrive in the form of a `std::vector<int>` or a `std::list<int>` or a `std::set<int>` or whatever.

I would like to take this time to point out (because everybody else is about to point this out) that the traditional way of doing this is to accept a pair of iterators. So make sure you have a two-iterator version. But you also want to make it more convenient to pass a container, too, because requiring people to pass a pair of iterators can be a hassle because you have to introduce a *name* and a *scope*.

```
extern std::set<int> get_the_ints();
// Convenient.
auto result = do_something_with(get_the_ints());

// Hassle.
auto the_ints = get_the_ints();
auto result = do_something_with(the_ints.begin(), the_ints.end());
```

Not only did you have to give a name to the set returned by `get_ the_ ints`, you now have to deal with the lifetime of that thing you just named. You probably want to destruct it right away, seeing as there’s no point hanging around to it, but that leaves you with some weird scoping issues.

```
{
  auto the_ints = get_the_ints();
  auto result = do_something_with(the_ints.begin(), the_ints.end());
} // destruct the_ints
// oops, I also lost the result!
```

If you wanted to accept anything and figure it out later, you could write

```
template<typename C>
auto do_something_with(C const& container)
{
  for (int v : container) { ... }
}
```

This takes anything at all, but if it's not something that can be used in a ranged for statement, or if the contents of the container are not convertible to `int`, you'll get a compiler error.

Maybe that's okay, but maybe the overly-generous version conflicts with other overloads you want to offer. For example, maybe you want to let people pass anything convertible to `int`, and you'll treat it as if it were a collection with a single element.

```
auto do_something_with(int v)
{
  ... use v ...
}
```

This overload looks fine, until somebody tries this:

```
do_something_with('x');
```

Now there is an ambiguous overload, because the `char` could match the first overload by taking `C = char`, or it could match the second overload via a conversion operator.

SFINAE to the rescue.

We can give the container version a second type parameter that uses SFINAE to verify that the thing is actually a container.

```
template<typename C, typename T = typename C::value_type>
auto do_something_with(C const& container)
{
  for (int v : container) { ... }
}
```

All standard containers have a member type named `value_type` which is the type of the thing inside the collection. We sniff for that type, and if no such type exists, then SFINAE kicks in, and that overload is removed from consideration, and we try the overload that looks for a conversion to `int`.

Now, it could be that you have a container that doesn't implement `value_ type`, but it still implements `begin` and `end` (presumably via ADL), so that the ranged for statement works. You can encode that in the SFINAE:

```
template<typename C,
    typename T = std::decay_t<
        decltype(*begin(std::declval<C>()))>>
auto do_something_with(C const& container)
{
  for (int v : container) { ... }
}
```

Starting with the type `C`, we use `std::declval` to pretend to create a value of that type, so that we can call `begin` on it, and then dereference the resulting iterator, and then decay it, producing a type `T` that represents the thing being enumerated. If any of these steps fails, say because there is no available `begin`, then the entire overload is discarded by SFINAE.

This was a bit of overkill because we never actually used the type `T`, but I kept it in because it sometimes comes in handy knowing what `T` is.

If you wanted to filter further to the case where the contents of the container are convertible to `int`, you can toss in some `enable_if` action:

```
template<typename C,
    typename T = std::decay_t<
        decltype(*begin(std::declval<C>()))>,
    typename = std::enable_if_t<
        std::is_convertible_v<T, int>>>
auto do_something_with(C const& container)
{
  for (int v : container) { ... }
}
```

Raymond Chen

Follow