Class template argument deduction may be the new hotness, but we'll always have maker functions

devblogs.microsoft.com/oldnewthing/20220420-00

April 20, 2022

Raymond Chen

C++17 introduced class template argument deduction, commonly known as CTAD. This is a core language feature that lets you omit class template arguments and let the compiler deduce them. For example:

```
// Old and busted
std::pair<int, int> p(1, 2);
auto p = std::pair<int, int>(1, 2);

// New hotness: compiler deduces <int, int>
std::pair p(1, 2);
auto p = std::pair(1, 2);
```

The catch with CTAD is that it's all-or-nothing. You either provide all of the template arguments, or you provide none of them. There is no going halfway.

```
// Doesn't work
std::pair<int> p(1, 2);
```

To be fair, CTAD is kind of forced into the all-or-nothing pattern because of the existence of template default arguments and variadic templates.

```
S<int> t(1, 2, 3);
```

Is that trying to construct a `S<int>` with three constructor parameters? Or is it trying to construct a `S<...>` where the first template argument is `int` and the compiler should try to deduce the others?[1]

This becomes a problem when you have a class where you'd like to deduce some but not all of the template arguments:

```
template<typename T, typename... Args>
struct S
{
    S(Args... args) { /* save the args somewhere */ }
    T get();
};

// Error: No matching constructor for S<double>
S<double> s(1, std::string("hi"));
```

We wished that line declared and constructed a `S<double, int, std::string>`, but it doesn't. What it declares is a `S<double>` and then tries to construct it with `(1, std::string("hi"))` as constructor parameters, which doesn't work.

But you know what *does* allow explicit specification of some template parameters with deduction of the others?

Functions, that's what.

Back in the old days, before CTAD, you had to use functions to do the things that CTAD does. That's why we have a bunch of maker functions like `std::make_pair`. Templated functions allow you to explicitly specify some template parameters, while allowing others to be deduced.

```
template<typename T, typename... Args>
auto make_s(Args... args)
{
    return S<T, Args...>(args...);
}

// Hooray, this creates a S<double, int, std::string>
auto s = make_s<double>(1, std::string("hi"));
```

Now that we have the basic idea, we can refine it. For one thing, our version above passes everything by value, which means that the string gets copied a lot. If we used universal forwarding references, that would avoid the extra copies. Here's our first attempt:

```
template<typename T, typename... Args>
auto make_s(Args&&... args)
{            // ^^
    return S<T, Args...>(args...);
}
```

This seems to work, but then you get use-after-free bugs when you try this:

```
auto make_s_double_int(int value)
{
    return make_s<double>(value);
}
```

Recall that forwarding references deduce a type parameter as an lvalue if the actual argument is an lvalue, and as a non-reference if the actual argument is an rvalue. In the case where you pass an lvalue as a parameter to `make_s`, the corresponding `Args` type is an lvalue reference.

In our case, it is an `int&`. That means that what we actually created was a `S<double, int&>`. And we passed a reference to a variable that is about to be destroyed as it goes out of scope.

What we need to do is remove the reference from the possibly-lvalue `Args` so that we create an `S<double, int>`, and then forward the arguments to the constructor.

```
template<typename T, typename... Args>
auto make_s(Args&&... args)
{
    return S<T, std::remove_reference_t<Args>...>
        (std::forward<Args>(args)...);
}
```

Okay, so that gets rid of one copy.

But there's another copy: The `S` constructor receives its parameters as values, not references, which means that the constructor will have to copy (or at least move) them into some local storage.[2] We could try to keep the parameters as references until the last moment.

```
template<typename T, typename... Args>
struct S
{
    template<typename... Actuals>
    S(Actuals&&... actuals) { /* save the args somewhere */ }
    T get();
};
```

But wait, we're not done yet. What if the parameter is a `const` lvalue?

```
const int x = 42;
auto s = make_s<double>(x);
```

We remove the reference, but we didn't remove the `const`, so this created a `S<double, int const>`. What we really want is the result you would get as if by assignment from the actual argument. And the name for that is `std::decay`.

```
template<typename T, typename... Args>
auto make_s(Args&&... args)
{
    return S<T, std::decay_t<Args>...>
        (std::forward<Args>(args)...);
}
```

All this will come in handy next time.

Bonus reading: Structured binding may be the new hotness, but we'll always have `std::tie`.

[1] I guess you could try to invent some new syntax to resolve the ambiguity, like `S<int, auto...>`, but you'd have to reconcile this against other uses of `auto` as a template argument, and actually just forget I even mentioned this.

[2] Return value optimization does not apply here because these aren't return values.

Raymond Chen

Follow