# Best Practices for Constructor Template Argument Deduction


## Mike Spertus

*Symantec, The University of Chicago*


## CTAD? Sounds complicated! What does it mean in simple terms?

**Constructor Template Argument Deduction (CTAD)** [1] is simply a fancy name for saying that C++17 will deduce class template arguments for constructors similarly to what C++ has always done with function templates. This can really simplify creating object from class templates, such as `vector`:

**Before C++17**

```
vector<int> v = {1, 2, 3}; // Why do I need to say this is a vector of ints?
lock_guard<shared_mutex> lck{smtx}; // Doesn't the compiler know the smtx is a shared_mutex?
shared_lock<shared_mutex> lck2{smtx2}; // Same here
scoped_lock<shared_mutex, shared_lock<shared_mutex>> assign_lock{smtx, lck2}; // Yuck!
```

By contrast, C++17 uses Constructor Template Argument Deduction to deduce the class template arguments of new objects without having to specify them explicitly:

**C++17**

```
vector v = {1, 2, 3}; // Deduces vector<int>, etc.
lock_guard lck{smtx};
shared_lock lck2{smtx2};
scoped_lock assign_lock{smtx, lck2};
```

## Wow! That looks useful and cool! How does it work?

In its simplest form, CTAD just treats every constructor in the primary class template (i.e., the main class template, specializations are not considered) as if it were a template function and then applies ordinary deduction:

```
template<typename T> struct A { A(T); };
A a(7);  // Deduces A<int>
```

Can it really be that simple? The original version [2] of the proposal back in 2007(!) proposed simply that. Unfortunately, six more versions of the proposal were needed to nail down the language specification!

One problem is that common cases lead to "non-deducible" template contexts. To see what can go wrong, suppose you want to initialize a vector from two iterators:

```
// b and e are iterators
vector v2(b, e); // How does the compiler know it is supposed to deduce the iterator's value_type?
```

After looking at examples like these (see the Best Practices for more examples), we realized that there needed to be some way for the programmer to guide template argument deduction when the default behavior was insufficient. Fortunately, C++17 provides a *deduction guide* facility to create custom deduction rules:

```
// Inside <vector>
template<typename Iter> // Explain how to deduce from iterator pair!
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>;
// Outside <vector>
vector v2(b, e); // Works now!
```

## Learning from the Standard Library

After we added CTAD to the core language, we naturally worked on updating the C++ standard library to leverage CTAD [3]. This not only helped improve the standard library and find/fix some issues with the core language feature [5, 6], but we also learned a lot about Best Practices for writing library classes to get the most from CTAD. In this poster, we share our experience from adding CTAD to the standard library to spare you from having to repeat all of the mistakes that we did on your way to learning the feature.

# Basic Best Practices

## When to use CTAD

Let's start at the beginning with the Best Practice for when Constructor Template Argument Deduction should be used in the first place. In C++14, constructor template argument deduction was often simulated via "factory functions" such as:

```
auto tup = make_tuple(1, 2.3); // Deduces tuple<int, double>
auto bi = back_inserter(v); // Deduces back_insert_iterator<decltype(v)>
```

Now that we have CTAD, should we get rid of such factory functions? It is certainly tempting, as they are inconsistently-named ugly boilerplate. Just constructing the class seems so much easier to write and clearer to use. Indeed, this is usually exactly what you should do. Factory functions like `back_inserter` and `make_boyer_moore_searcher` don't really serve any purpose in C++17, and `make_boyer_moore_searcher` was removed from the standard library when the string search utilities were incorporated into C++17 from the Library Fundamentals TS.

However, there are cases where you should consider sticking with a factory function. Consider `make_tuple`, which does more than just call the tuple constructor, it also unwraps `reference_wrapper` arguments into references. While it is easy enough to write a deduction guide to emulate this behavior, this could silently lead to unexpected behavior since CTAD looks exactly like a constructor call. Therefore, our Best Practice is to only use CTAD for straightforward deduction of template arguments and use factory functions to call out special behavior: `make_tuple` should be retained as a factory function.

## Understand Copying vs Wrapping

For classes that can be constructed from an initializer list, it can be confusing whether a braced initializer with a single element means copying with the copy constructor or wrapping with the initializer list constructor:

```
vector v{1, 2, 3, 4}; // Deduces vector<int>
vector v2{v, v, v}; // Deduces vector<vector<int>>
vector v3{v}; // What is v3? vector<int>? vector<vector<int>>
```

Indeed, this is not so obvious, but after extensive committee discussion [6], consensus was reached that copying should always take preference over wrapping when both are possible, so `v3` will be a `vector<int>`.

While this is usually what you want, you can still force list initialization (e.g. in generic code) by invoking your constructor in a way that could not possibly be a copy, such as:

```
vector v4({v}); // Make clear you want vector<vector<int>>
```

## Put parameter packs at end of argument lists

A funny thing happened when we tried using Constructor Template Argument Deduction with `scoped_lock`, which had a constructor that takes a parameter pack of mutex types followed by an `adopt_lock_t`. This is perfectly legal for class templates, but in a function template, the variadic parameter pack needs to be the last argument for deduction to work. Since CTAD generates function templates from class template constructors, this can result in function templates with awkwardly placed parameter packs:

```
mutex m1, m2;
scoped_lock l{m1, m2, adopt_lock}; // Oops, adopt_lock is interpreted as a mutex
```

The solution is simple. We just fixed `scoped_lock` to take the `adopt_lock_t` before the parameter pack:

```
scoped_lock l{adopt_lock, m1, m2}; // Works with the C++17 version of scoped_lock
```

# Advanced Best Practices

## Make your constructors deducible

Different ways of writing classes that behave the same in C++14 can behave differently in C++17! For example, the standard defines `vector` like:

```
template<class T, class Allocator = allocator<T>>
struct vector { vector(initializer_list<T>, Allocator = Allocator()); /* ... */ };
/* ... */
vector v = {1, 2, 3}; // Correctly deduces vector<int>
```

However, suppose a compiler implemented `vector` something like:

```
template<class T, class Allocator = allocator<T>>
struct vector {
  using value_type = Allocator::value_type;
  vector(initializer_list<value_type>, Allocator); /* ... */ };
/* ... */
vector v = {1, 2, 3}; // Oops, no way to deduce T!
```

In C++14, these two definitions are equivalent, so this was allowed by the *as-if* rule, but when doing CTAD they are different because only knowing `Allocator::value_type` in no way tells you what `T` is (especially because we don't even know what `Allocator` is), leading to a *non-deducible context*.

Note that the indirection through `Allocator` is what causes the problem, as CTAD has special language to deduce from direct aliases like "`using value_type = T;`"

This example could easily have been avoided by simply writing `vector` without the redundant indirection in the first place in accordance with this best practice. However, what do you do if you have legacy classes that can't be rewritten or constructors that are intrinsically non-deducible, like constructing a `vector` from two iterators? Such cases can still be handled by explicitly writing deduction guides

```
template<class T, class Allocator = allocator<T>>
vector(initializer_list<T>, Allocator = Allocator()) -> vector<T, Allocator>;
template<typename Iter> vector(Iter b, Iter e) -> vector<typename Iter::value_type>;
// Outside <vector>
vector v2{b, e}; // Works now!
```

## Consider passing arguments by value

While CTAD is carefully designed to work well with both reference and value arguments [7], passing arguments by value can avoid a few gotchas. To see a good illustration of this, let's look at `pair` from the standard library. `pair<T, U>` has a constructor `pair(T const &, U const &)`. While this usually works well, the failure to decay could lead to a few surprises:

```
// What would CTAD for pair look like without deduction guides?
pair p{7, make_shared<string>("foo")}; // Nearly everything works great, like in this example
// But there are problems for types that can be "decayed"
void f() {}
int a[5];
pair p2{a, 3}; // Deduces pair<int[5], int>, where pair<int *, int> would probably be better
pair p3{f, 3}; // Deduces pair<void(), int>, which doesn't make sense
```

There are several choices for how to handle this.

- The first is simply to ignore it. How often do you need array-to-pointer and function-to-pointer conversion anyway ? Even if it happens, you can always explicitly give the template arguments if deduction fails (Just like in C++14)
- Have your constructor take arguments by value. While doing this when inappropriate would be the tail wagging the dog, in many cases, either by-value or by-reference is reasonable. In those cases, consider taking constructor arguments by value
- If the constructor takes a reference, creating an equivalent "by-value" deduction guide will force decay, which is exactly what the standard library does with `pair`:

```
template<class T, class U> pair(T, U) -> pair<T, U>;
```

# My Favorite Best Practice

## Get ready for concepts by constraining your class templates

While this Best Practice is a little more abstract than the previous ones, it is my favorite because it is a new Best Practice that is showing up in multiple language features, suggesting a deeper unifying meaning that tells you that you are on the right track. More concretely, we had to put many constrained deduction guides in `unordered_set` to avoid constructor ambiguity. For example, it is unclear in an expression like `unordered_set(5, x)` whether you mean to invoke `unordered_set(size_type, Hash)` or `unordered_set(size_type, Allocator)`.

Reflecting on the ambiguity, the real point here is that since `Hash` and `Allocator` are not constrained, they are just names. If only the compiler understood that Allocator is an allocator and Hash is a hashing function, there would be no ambiguity. But this is also one of the main ideas behind Concepts!

To make the point more clearly, imagine that there were `is_allocator_v` and `is_hash_v` type traits. Suppose we then added constraints to the definition of `unordered_set` as follows

```
template<class Key,
  class Hash = hash<Key>, class KeyEqual = equal_to<Key>, class Alloc = allocator<Key>,  // Usual
  typename = enable_if_t<is_hash_v<Hash>>, typename = enable_if_t<is_allocator_v<Alloc>>> // New constraints
> class unordered_set { /* ... */ };
```

Now, there is no longer a need to write a deduction guide to select the right constructor. It is right there in the constraints!

Of course, once Concepts become part of C++, you will use concepts and the above becomes

```
template<class Key, class Hash = hash<Key>, class KeyEqual = equal_to<Key>, class Alloc = allocator<Key>>
  requires is_hash_v<Hash> && is_allocator_v<Alloc>
class unordered_set { /* ... */ };
```

In fact, if you use a Concepts-enabled compiler today, code like the above correctly guides Constructor Template Argument Deduction without needing to manually tweak with deduction guides. You can run code demonstrating this at the QR code above.

By following the new Best Practice of constraining your template classes, not only will Constructor Template Argument Deduction work better today, your code will be positioned to take advantage of Concepts in the future

# References


1. Spertus, Vali, Smith (2016), *Template argument deduction for class templates*, P0091R3, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0091r3.html
2. Spertus (2007), *Argument Deduction for Constructors*, WG21/N2332=07-0192, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2332.pdf
3. Spertus, Brown, Lavavej (2017), *Toward a resolution of US7 and US14: Integrating template deduction for class templates into the standard library*, P0433R2, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0433r2.html
4. Spertus, Brown, Lavavej (2017), Some improvements to class template argument deduction integration into the standard library, P0739R0, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0739r0.html
5. Merrill (2017), *Drafting for class template argument deduction issues*, P0620R0, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0620r0.html
6. Spertus, Merrill (2017), *Language support for Class Template Argument Deduction* , P0702R1, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0702r1.html
7. Spertus, Smith, Vali (2016), *Class Template Argument Deduction: Assorted NB resolution and issues*, P0512R0, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0512r0.pdf


See Sample Code Live in Wandbox!