Class Template Argument Deduction for Everyone

Stephan T. Lavavej ("Steh-fin Lah-wah-wade")
Principal Software Engineer, Visual C++ Libraries
stl@microsoft.com
@StephanTLavavej

Getting Started

- Please hold your questions until the end
  - Write down the slide numbers
- Everything here is Standard
- **CTAD** = **C**lass **T**emplate **A**rgument **D**eduction
- CTAD is available in VS 2017 15.7
  - Compile with `/std:c++17`
  - Otherwise you'll get errors like:
    - `error C2955: 'std::pair': use of class template requires template argument list`

Why Examples Use STL Types

- The STL is:
  - Near-universally used, so hopefully it's familiar
  - The first library to completely support CTAD
  - Sufficiently complicated to illustrate interesting scenarios
  - What I work on all day, every day
- CTAD is a Core Language feature
  - Not restricted to the STL
- Look for opportunities to use CTAD everywhere
  - Class templates in Boost, other libraries, your own code

Templates before C++17

"You enter what seems to be
an older, more primitive world."

Function Template Arg Deduction

- equal(InIt1, InIt1, InIt2, InIt2)
  - InIt1 and InIt2 are parameters; need concrete arguments
  - Function call: equal(s.begin(), s.end(), ptr, ptr + n)
  - The compiler knows the types of s.begin() etc.
  - InIt1 is deduced to be set<int>::iterator
  - InIt2 is deduced to be const long long *
- Template argument deduction: complex rules work well
  - Makes function templates far more usable!
  - lst.begin(), vec.end() deduction failure: inconsistent
- Explicit template arguments: don't help the compiler
  - make_shared<int>() good, swap<int, int>() bad

Class Templates: No Deduction

- `pair<int, int> p(11, 22);`
  - The compiler knew that `11` was `int`, but wouldn't help you
  - `int` is my usual example; real types are obnoxiously long
  - Verbose repetition is a chance to make mistakes
- Workaround: `make_pair(11, 22)`
  - More template machinery: forwarding and decay
  - Reduced compiler throughput: front-end and back-end
  - Slow, bloated non-optimized codegen: actual function calls
  - Annoying to debug: stepping through helper functions
  - Still verbose: `make_` prefix, `auto` for variables

CTAD Examples

For Everyday Code

Hello, CTAD World

```
#include <type_traits>
#include <utility>
using namespace std;
int main() {
  pair p(1729, "taxicab");
  static_assert(is_same_v<
    decltype(p), pair<int, const char *>>);
}
```

CTAD Syntax

- Works with parentheses and braces:

```
pair x(11, 22);
pair y{11, 22};
```

- Works with direct-init and copy-init:

```
pair z = y;
pair a = { 11, 22 };
```

- Works with named variables and temporaries:

```
vector<UserDefinedType> v;
v.emplace_back(pair(11, 22));
v.emplace_back(pair{11, 22});
```

What CTAD Is Doing

- C++17 performs template argument deduction
  - When constructing an object
  - Given only the name of a class template
  - Constructor arguments provide type information
- Usual deduction rules and library authors control what types are deduced
  - `1729` is an rvalue of type `int`
  - `"taxicab"` is an lvalue of type `const char [8]`
  - The STL deduces `pair<int, const char *>`
  - We'll see how this works later (deduction guides)

Lock Guards, 1 of 3

```
shared_mutex cat_mutex;

{
  shared_lock reading(cat_mutex);
  observe(cat);
}
// shared_lock<shared_mutex>
```

Lock Guards, 2 of 3

```
shared_mutex cat_mutex;

{
  lock_guard writing(cat_mutex);
  modify(cat);
}
// lock_guard<shared_mutex>
```

Lock Guards, 3 of 3

```
shared_mutex cat_mutex;
mutex dog_mutex;
{
  scoped_lock writing2(cat_mutex, dog_mutex);
  modify(cat, dog);
}
// scoped_lock<shared_mutex, mutex>
```

Reverse Sort

```
array arr = { "lion"sv, "direwolf"sv,
  "stag"sv, "dragon"sv };
sort(arr.begin(), arr.end(), greater{});
cout << arr.size() << ": ";
for (const auto& e : arr) { cout << e << " "; }
cout << "\n";
// 4: stag lion dragon direwolf
```

array CTAD Is Awesome

```
array arr = { "lion"sv, "direwolf"sv,
  "stag"sv, "dragon"sv };
```

- arr is `array<string_view, 4>`
  - Both element type and size are deduced (by a guide)
- `greater{}` is an object of type `greater<void>`
  - CTAD works with default template arguments
  - `template <typename T = void> struct greater;`
  - Slightly less verbose than `greater<>{}`
  - Still need `greater<>` when forming types, not objects

Range Construction

```
list<pair<int, string>> lst = {
  { 1966, "TOS" }, { 1987, "TNG" },
  { 1993, "DS9" }, { 1995, "VOY" } };
vector vec(lst.rbegin(), lst.rend());
for (const auto& [i, s] : vec) {
  cout << i << " " << s << "; "; }
cout << "\n";
// 1995 VOY; 1993 DS9; 1987 TNG; 1966 TOS;
```

Type Transformation

```
list<pair<int, string>> lst = { /*...*/ };
vector vec(lst.rbegin(), lst.rend());
```

- Arg: `list<pair<int, string>>::reverse_iterator`
- CTAD: `vector<pair<int, string>>`
- The deduction guide for the range constructor transforms an iterator type into an element type

CTAD Errors

For Everyday Code

pair of Nothing

```
meow.cpp(5,10):  error: no viable constructor
or deduction guide for deduction of template
arguments of 'pair'
    pair p{};
         ^
```

- pair doesn't have default template arguments
  - Unlike `greater`
- The compiler won't guess what you want from any following code; info almost never flows backwards

array<Inconsistent, N>

```
array arr = { 11, 22, 3.14 };

[...]\array(233,2):  error: static_assert failed due
to requirement 'conjunction_v<is_same<int, int>,
is_same<int, double> >' "N4687 26.3.7.2
[array.cons]/2: Requires: (is_same_v<T, U> && ...)
is true. Otherwise the program is ill-formed."
[...]
```

shared_ptr to Scalar or Array

```
meow.cpp(6,16):  error: no viable constructor
or deduction guide for deduction of template
arguments of 'shared_ptr'
    shared_ptr sp(new string);
               ^
```

- new string and new string[5] are string *
  - shared_ptr<string> or shared_ptr<string[]>?
- unique_ptr behaves identically
- Use make_shared() and make_unique()

CTAD Limitations

For Everyday Code

C++17 CTAD Limitations

- P1021 by Mike Spertus and Timur Doumler proposes fixing several C++17 CTAD limitations for C++20
- CTAD currently doesn't work with:
  - Alias templates
  - Explicit template arguments
- CTAD currently needs deduction guides for:
  - Aggregates (like `std::array`)
  - Inherited constructors

Alias Templates

```
template <typename K, typename A = allocator<K>>
  using reverse_set = set<K, greater<>, A>;

meow.cpp(10,17):  error: alias template
'reverse_set' requires template arguments; argument
deduction only allowed for class templates
    reverse_set r = { 11, 22, 33 };
                ^
```

Explicit Template Arguments

- CTAD is currently all-or-none:

```
meow.cpp(6,5):  error: too few template
arguments for class template 'array'
    array<string> a = { "Runaway", "Robots" };
    ^
```

Automatic CTAD

For Library Code

Non-Templated Constructors

```
template <typename A, typename B>
  struct MyPair {
  MyPair(const A&, const B&) { }
};
MyPair mp1{1729, 3.14}; // MyPair<int, double>
MyPair mp2{22, "meow"}; // MyPair<int, char[5]>
```

- Works automatically without deduction guides
  - If you don't want `MyPair<int, char[5]>`, you'll need guides

Constructible != Deducible

```
template <typename A, typename B>
  struct Peppermint {
  explicit Peppermint(const A&) { }
};
Peppermint<int, double> p1{11};
Peppermint p2{22};
```

Helpful Compiler Error

```
meow.cpp(10,16):  error: no viable constructor or deduction
guide for deduction of template arguments of 'Peppermint'
    Peppermint p2{22};
               ^
meow.cpp(5,14):  note: candidate template ignored: couldn't
infer template argument 'B'
    explicit Peppermint(const A&) { }
             ^
meow.cpp(4,42):  note: candidate template ignored: could not
match 'Peppermint<A, B>' against 'int'
template <typename A, typename B> struct Peppermint {
                                         ^
```

Constructible != Deducible, Again

```
template <typename A, typename B> struct Jazz {
  Jazz(A *, B *) { }
};
int i = 1729;
const double d = 3.14;
Jazz j1{&i, &d}; // Jazz<int, const double>
Jazz<int, int> j2{&i, nullptr};
Jazz j3{&i, nullptr}; // note: candidate template
ignored: could not match 'B *' against 'nullptr_t'
```

Default Template Arguments

```
template <typename T,
  typename Alloc = allocator<T>>
  struct Spot {
  explicit Spot(const T&) { }
  Spot(const T&, const Alloc&) { }
};
struct MyAlloc { };
Spot s1{11}; // Spot<int, allocator<int>>
Spot s2{22, MyAlloc{}}; // Spot<int, MyAlloc>
```

Will CTAD Automatically Work?

- CTAD automatically works when:
  - A class template has a constructor whose signature mentions all of the class template's parameters
    - `MyPair<A, B>` had `MyPair(const A&, const B&)`
  - Or the class template provides default template arguments
    - `Spot<T, Alloc>` had `Spot(const T&)` and `Alloc = allocator<T>`
- CTAD doesn't automatically work when:
  - Class template parameters aren't mentioned/defaulted
  - Arguments prevent deduction (e.g. `nullptr` arg for `B *`)
  - Parameters are non-deducible (e.g. `list<T>::iterator`)

Deduction Guides

For Library Code

Range Constructors

```
template <typename T> struct MyVec {
  template <typename It> MyVec(It, It) { }
};
```

- This breaks the connection that CTAD relies on
  - CTAD can still work for other constructors, but not this one
- It's time to help the compiler
  - By providing a deduction guide

Hello, Deduction Guide World

```
template <typename T> struct MyVec {
  template <typename It> MyVec(It, It) { }
};
template <typename It> MyVec(It, It)
  -> MyVec<typename iterator_traits<It>
              ::value_type>;
int * ptr = nullptr;
MyVec v(ptr, ptr); // MyVec<int>
```

Forwarding Constructors

```
template <typename A, typename B>
  struct AdvancedPair {
  template <typename T, typename U>
    AdvancedPair(T&&, U&&) { }
};
```

- Perfect forwarding inhibits CTAD
- We want to imitate `make_pair()` which decays
- Note: `make_pair()` unwraps `reference_wrapper`
  - `std::pair` CTAD doesn't; `reference_wrapper` is rare

Decay by Value (IMPORTANT!)

```
template <typename A, typename B>
  struct AdvancedPair {
  template <typename T, typename U>
    AdvancedPair(T&&, U&&) { }
};
template <typename X, typename Y>
  AdvancedPair(X, Y) -> AdvancedPair<X, Y>;
AdvancedPair adv(1729, "taxicab");
// AdvancedPair<int, const char *>
```

How "Decay by Value" Works

```
template <typename X, typename Y>
  AdvancedPair(X, Y) -> AdvancedPair<X, Y>;
```

- Tells CTAD to perform template argument deduction for a hypothetical signature `AdvancedPair(X, Y)` taking arguments by value
  - Such deduction performs decay
  - If it succeeds, CTAD produces `AdvancedPair<X, Y>`
- `decay_t` would also work, but would be verbose

What "Decay by Value" Illustrates

- CTAD determines what type to construct
  - Input: constructor args, constructors, deduction guides
  - Performs template arg deduction and overload resolution
  - Output: a specific type (or deduction failure)
- Next, overload resolution happens again, normally
- **CTAD doesn't affect the constructor call**
  - CTAD and deduction guides don't affect existing code
- Example:
  - CTAD uses `AdvancedPair(X, Y)`
  - CTAD deduces `AdvancedPair<int, const char *>`
  - Overload resolution selects `AdvancedPair(T&&, U&&)`

Enforcing Requirements

```
template <typename T, size_t N> struct MyArray {
  T m_array[N];
};
template <typename First, typename... Rest>
  struct EnforceSame {
  static_assert(conjunction_v<is_same<First, Rest>...>);
  using type = First;
};
template <typename First, typename... Rest>
  MyArray(First, Rest...) -> MyArray<typename EnforceSame<
    First, Rest...>::type, 1 + sizeof...(Rest)>;
MyArray a = { 11, 22, 33 }; // MyArray<int, 3>
```

Supporting CTAD in Your Library

- For each constructor of each class template:
  - Is CTAD applicable? (Example "no": `vector<T>(size_t)`)
  - Does CTAD automatically work?
    - And does it do what you want? (Recall `MyPair<int, char[5]>`)
  - Or do you need to write a deduction guide?
- If you support pre-C++17, guard your guides
  - Feature-test macro: `__cpp_deduction_guides`
- Write simple tests with `static_assert`
  - Preventing constructors/typedefs from breaking things

Corner Cases

For Experts

Explore, Don't Imitate

- CTAD can be complicated
  - Template argument deduction can be complicated
  - Overload resolution can be complicated
  - CTAD involves both
- Let's explore corner cases to learn:
  - How to avoid them
  - How to deal with them
  - How CTAD works in ordinary cases
- Don't imitate the following weird code

Non-Deduced Contexts

```
template <typename X> struct Identity {
  using type = X;
};
template <typename T> struct Corner1 {
  Corner1(typename Identity<T>::type, int) { }
};
Corner1 corner1(3.14, 1729);
meow.cpp(6,3):  note: candidate template ignored:
couldn't infer template argument 'T'
  Corner1(typename Identity<T>::type, int) { }
  ^
```

- Avoid non-deduced contexts, or provide deduction guides

Non-Deduced Contexts, Again

```
template <typename X> struct Identity {
  using type = X;
};
template <typename T> struct Corner1a {
  using U = typename Identity<T>::type;
  Corner1a(U, int) { }
};
Corner1a corner1a(3.14, 1729);
meow.cpp(7,3):  note: candidate template ignored:
couldn't infer template argument 'T'
```

- Nested typedefs don't fix non-deduced contexts

Nested Typedefs Aren't Bad

```
template <typename T> struct Purr {
  using U = T;
  Purr(U, int) { }
};
Purr purr(3.14, 1729); // Purr<double>
```

- MSVC's STL extensively uses nested typedefs
  - None of them needed to change for CTAD
  - Your code may vary

More Info

More Info

- VS 2017 15.6: CTAD (STL)
  - Reported two compiler bugs in Clang 5, fixed by Clang 6
- VS 2017 15.7: CTAD (C1XX, EDG)
- VS 2017 15.8: Feature-test macros
  - `__cpp_deduction_guides`
- C++20 WP: https://wg21.link/standard
- Core papers: P0091, P0512, P0620, P0702
- Library papers: P0433, P0739

Questions?

stl@microsoft.com

@StephanTLavavej

Bonus Slides!

More Corner Cases

For Experts

tuple/optional Avoid Wrapping

```
tuple tup{11, 22, 33}; // tuple<int, int, int>
tuple tup2{tup};       // tuple<int, int, int>

optional opt{"meow"s}; // optional<string>
optional opt2{opt};    // optional<string>
```

- CTAD prioritizes everyday code
- Expert library authors need to be aware of this
  - If they want `tuple<tuple<Args>>`, `optional<optional<T>>`

initializer_list Overloading

- Examples mentioned by Nicolai Josuttis

```
set<int> s;
vector v1(s.begin(), s.end()); // vector<int>
vector v2{s.begin(), s.end()};
  // vector<set<int>::iterator>

vector v3{"ab", "cd"}; // vector<const char *>
vector v4("ab", "cd"); // vector<char>, UB!
```

- Unlikely to be problematic in everyday code

CTAD Before Overload Resolution

```
template <typename X> struct Identity {
  using type = X;
};
template <typename T> struct Corner2 {
  Corner2(T, long) { }
  Corner2(typename Identity<T>::type, unsigned long) { }
};
Corner2 corner2(3.14, 1729);
meow.cpp(11,11):  error: call to constructor of
'Corner2<double>' is ambiguous
```

- CTAD succeeds, but overload resolution fails

What Happened?

```
Corner2(T, long);
Corner2(typename Identity<T>::type, unsigned long);
Corner2 corner2(3.14, 1729);
```

- CTAD uses `(T, long)`, ignores non-deduced context
  - Success: `T` is `double`; `1729` is convertible to `long`
  - We're constructing `Corner2<double>`
- Overload resolution for arguments `double, int`
  - `(double, long)` vs. `(double, unsigned long)`
  - Ambiguous because neither integral conversion is preferred!

How Can We Avoid That?

```
Corner2(T, long);
Corner2(typename Identity<T>::type, unsigned long);
Corner2 corner2(3.14, 1729);
```

- What's to blame?
  - CTAD's rules?
  - The non-deduced context?
  - The constructor arguments?
- None of the above – this is a bad overload set
  - `Corner2<double>{3.14, 1729}` was already ambiguous
  - The non-deduced context isn't the root cause here

Deduction Guides, 1 of 3

```
template <typename T> struct Corner3 {
  Corner3(T) { }
  template <typename U> Corner3(U) { }
};
```

```
Corner3 corner3(1729);
```

- `Corner3<int>` without deduction guide
- …

Deduction Guides, 2 of 3

```
template <typename T> struct Corner3 {
  Corner3(T) { }
  template <typename U> Corner3(U) { }
};
template <typename X> Corner3(X)
  -> Corner3<X *>;
Corner3 corner3(1729);
```

- `Corner3<int>` without deduction guide
- `Corner3<int *>` with deduction guide

Deduction Guides, 3 of 3

- Deduction candidates: hypothetical function templates
  - Generated from ctors, guides, and "copy deduction candidate"
    - N4762 11.3.1.8 [over.match.class.deduct]/1
  - CTAD performs template argument deduction and overload resolution for these deduction candidates
  - Overload resolution still prefers "more specialized"
    - N4762 11.3.3 [over.match.best]/1.7
  - If equally specialized, new tiebreaker: prefer guides
    - N4762 11.3.3 [over.match.best]/1.12
- `Corner3(T)` ctor, `Corner3(X)` guide
  - Equally specialized, so the guide wins

basic_string CTAD Ambiguity

```
// https://wg21.link/lwg3076
string s0;
basic_string s1(s0, 1, 1);
// WANT: basic_string(const basic_string&, size_type,
//    size_type, const Allocator& = Allocator())
// CONFLICT: basic_string(size_type, charT,
//    const Allocator&)

basic_string s3("cat", 1);
// WANT: basic_string(const charT *, size_type,
//    const Allocator& = Allocator())
// CONFLICT: basic_string(const charT *, const Allocator&)
```

Fixing `basic_string` CTAD

- `basic_string`'s ctors are too heavily overloaded
- In these cases, CTAD deduced `Allocator = int`
- Deduction guides can't easily fix this scenario
- Fix: constrain `Allocator` to be an allocator
  - Avoids affecting actual construction
  - Probably superseded by concepts

More Automatic CTAD

For Library Code

Constrained Constructors

```
template <typename A, typename B> struct Garfield {
  Garfield() { }
  template <typename A2 = A,
    enable_if_t<!is_same_v<A2, B>, int> = 0>
    Garfield(A, B) { }
};
Garfield g1{1729, 3.14}; // Garfield<int, double>
Garfield g2{22, 22}; // note: candidate template
ignored: requirement '!is_same_v<int, int>' was
not satisfied [with A = int, B = int, A2 = int]
```

- Templated constructors can work with CTAD

<charconv>

C++17's Final Boss

C++17 <charconv> Overview

- `from_chars()` and `to_chars()`
- Integer and floating-point
- Low-level: no whitespace, no locales, few options
- Bounds-checked
- No null termination for input or output
- No dynamic memory allocation
- No exceptions
- Potentially amazing performance

Implementation Progress

- VS 2017 15.7: Integer `from_chars()`/`to_chars()`
  - Not yet hyper-optimized
- VS 2017 15.8: Floating-point `from_chars()`
  - Derived from UCRT `strtod()`/`strtof()`, ~40% faster
- VS 2017 15.9: Floating-point `to_chars()`, partially
  - Shortest round-trip decimal overloads
  - Powered by Ulf Adams' new algorithm, Ryu
- VS 2019 16.0: ~60-80% faster `fixed` notation
  - Elementary school long division, suggested by Ulf Adams

Implementation Progress

- Currently 199 KB of source code in headers
  - 11 KB of `constexpr` lookup tables for generated code
- Currently 264 KB of tests
  - Mostly due to voluminous test data
- Remaining implementation work:
  - `to_chars()` hexfloat shortest round-trip
  - `to_chars()` hexfloat precision
  - `to_chars()` decimal precision
  - More optimization work? (Integer, `fixed`, C2, LLVM)

Perf: scientific vs. %.8e %.16e

Type	to_chars()	sprintf_s()	Speedup	Platform
float	53.3 ns	602.1 ns	11.3x	x86 C2
float	46.8 ns	587.7 ns	12.6x	x86 LLVM
double	110.2 ns	2739.2 ns	24.9x	x86 C2
double	79.7 ns	2727.4 ns	34.2x	x86 LLVM
float	43.5 ns	418.1 ns	9.6x	x64 C2
float	37.5 ns	415.5 ns	11.1x	x64 LLVM
double	54.3 ns	1708.1 ns	31.4x	x64 C2
double	46.6 ns	1707.3 ns	36.6x	x64 LLVM

Perf: `fixed` (lossless) vs. %f (lossy)

Type	`to_chars()`	`sprintf_s()`	Speedup	Platform
`float`	76.3 ns	552.4 ns	7.2x	x86 C2
`float`	69.5 ns	556.2 ns	8.0x	x86 LLVM
`double`	690.2 ns	2687.3 ns	3.9x	x86 C2
`double`	764.1 ns	2688.8 ns	3.5x	x86 LLVM
`float`	61.0 ns	399.5 ns	6.5x	x64 C2
`float`	51.2 ns	400.8 ns	7.8x	x64 LLVM
`double`	419.8 ns	1702.6 ns	4.1x	x64 C2
`double`	334.2 ns	1720.2 ns	5.1x	x64 LLVM