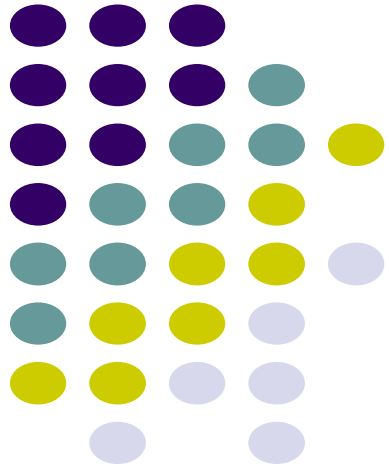
Inference: The Big Picture

September 18, 2019

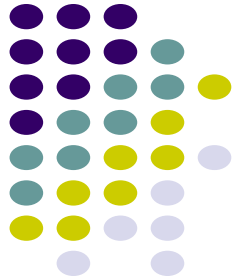
Mike Spertus

Symantec, University of Chicago

mike@spertus.com

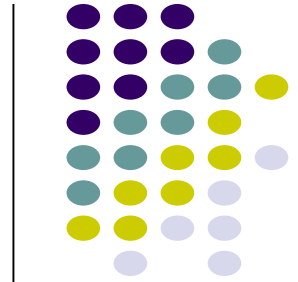


The official topic of this talk
Inference

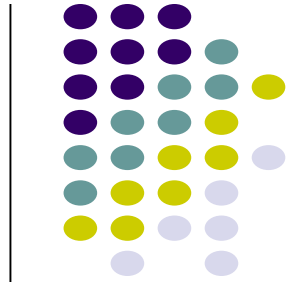


- C++' ability "infer" a type (or occasionally a value) without the need to explicitly specify it
- Example: No need to specify template arguments for the `sort` function template below
  - `sort(myVec.begin(), myVec.end());`
- `sort<vector<int>::iterator>` is inferred
- Many more details to come…

The real topic of this talk

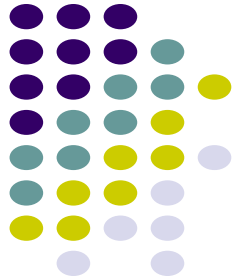


> Generic Programming Should "Just" Be Normal Programming
>
> — Bjarne Stroustrup

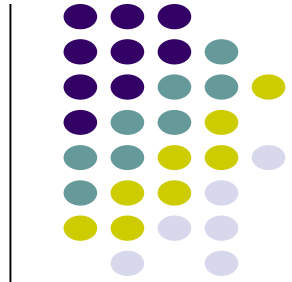


WHAT’S THE CONNECTION?

How inference makes generic programming like normal programming

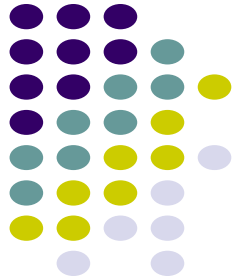


- Inference is what lets us call the function template `sort` as if it were a normal function
  - `sort(myVec.begin(), myVec.end());`
- Or create the class template `scoped_lock` as if it were a normal class
  - `scoped_lock lck(myMutex);`
- Inference is letting use function templates and class templates as if they were normal functions and classes!
- In other words, inference is a way of making template programming like normal programming



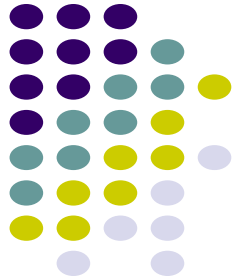
WHY SHOULD GENERIC PROGRAMMING BE LIKE NORMAL PROGRAMMING?

Solving for the right problem



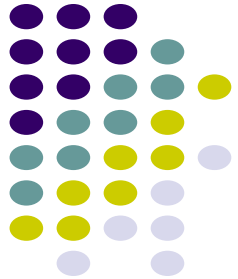
- Before we talk about *how* to make generic programming like normal programming, we need to understand *why* generic programming should be like normal programming
  - Upcoming paper with Bjarne Stroustrup
- Indeed, it is a natural reaction to think that template code should be visibly different from normal code

Reducing boilerplate



- People often think of inference primarily as a convenience for writing code with less boilerplate
  - e.g., so we don't have to specify the template parameter to `sort`
- Indeed, the lack of repetitive boilerplate in C++ is one of the reasons we wake up every morning glad we aren't Java programmers ☺

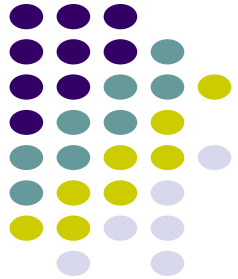
In fact, reduction in boilerplate can be very worthwhile

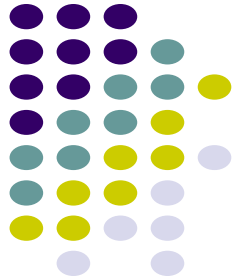
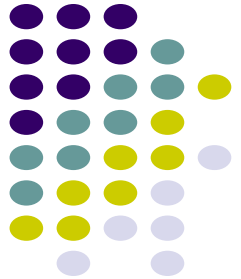
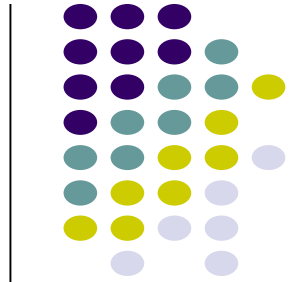
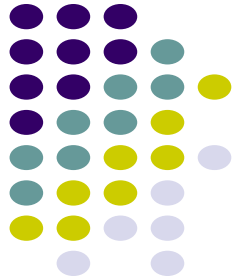
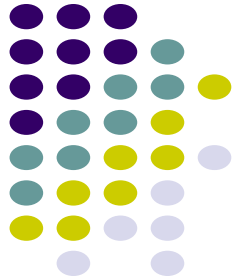
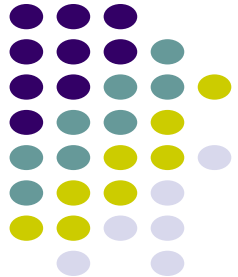
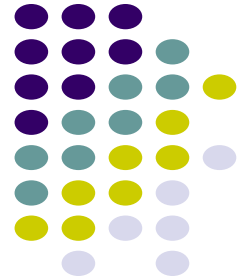
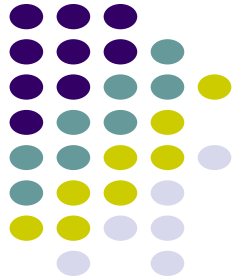
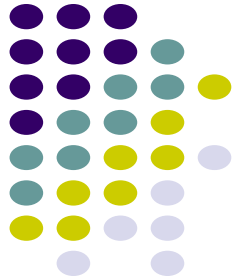
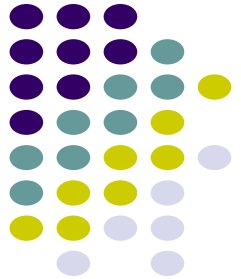
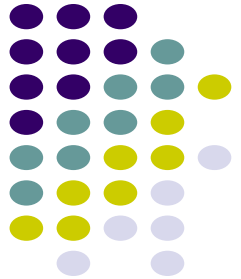
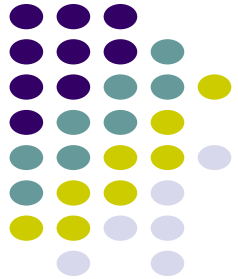
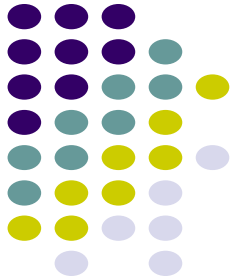
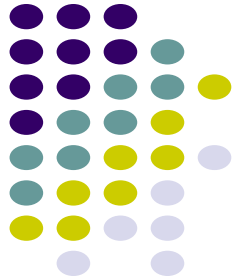
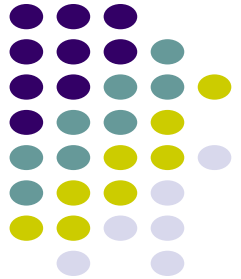
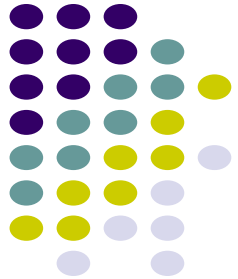
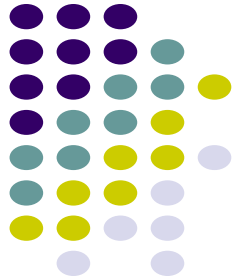
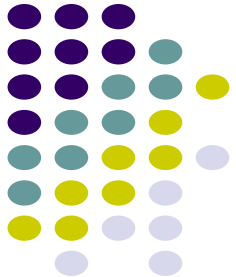


- For example, here's how p0429's flat_map might be created in C++14

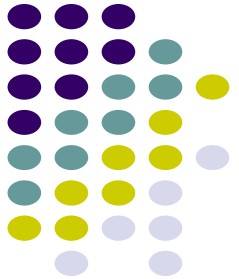
```
template<class Cont1, class Cont2>
 f(Cont1 &c1, Cont2 &c2) {
   flat_map
       <typename Cont1::value_type,
        typename Cont2::value_type,
        less<typename Cont1::value_type>,
        Cont1, Cont2>
   fm(c1, c2);
   /* ... */
 }
```

The same code with inference (C++17 CTAD)



- ```
  template<class Cont1, class Cont2>
  f(Cont1 &c1, Cont2 &c2) {
    flat_map fm(c1, c2);
    /* ... */
  }
  ```
- Obviously, the C++17 version gives us
  - Much less boilerplate
  - Much better signal-to-noise
  - Code much more like “normal programming”
  ```

The real point of inference: Improve maintainability



- Nice to have
  - Reducing repetitive boilerplate
- Transformative
  - Inference makes code less brittle and less tightly-coupled
- To see why, we will consider why programs become brittle

Brittleness



- A program shows great progress in the first couple of iterations
- Changing version 10 is dangerous
  - The code inevitably becomes brittle
    - they’re lucky if they can simply change a font without breaking something!
- Usual reason is “tightly coupled” code
  - Changing one thing requires changing many other parts of the program that are “coupled” to the change



TEMPLATES VS NORMAL IS ONE OF THE MOST TIGHTLY-COUPLED DECISIONS IN PROGRAMMING

The Point



- C++ has two different models for extensibility and type-generic algorithms
  - “Normal/OO Programming”
    - Runtime-Dispatch, polymorphism
  - “Generic/Template programming”
    - Compile-time Dispatch

This is Good



- A strength of C++
  - the programmer can choose between runtime and compile-time dispatch
    - OO for simplicity and flexibility
    - Templates for performance
- Robust support for both paradigms is an important part of why C++ is so successful

This is also Bad



- In traditional C++, normal programming looks very different from generic programming
- Programmer has to choose whether to use OO or templates for extensible type-generic programming right from the get go
  - The tradeoffs are complicated
  - The tradeoffs may change as time goes on
- Once that decision is made, it is virtually impossible for the programmer to change their mind
  - Programs become brittle and unmaintainable



MOTIVATING EXAMPLES

Example 1: Turning a base class into a concept



- Typical graphics code (`Curve` is a base class)
-
```
pair<Curve, vector<Point<double>>>
f(Curve &c)
{
  Point<double> top_left{c.x_min(), c.y_max()};
  screen->line(top_left, {c.x_min(), c.y_max()});
  vector<Point<double>>
    cp(c.ctrl_pts.begin(), c.ctrl_pts.end());
  return {c, cp};
};
```

- Later we may wish we had used templates instead of virtuals for better performance
  - Traditional C++: We're stuck
  - Modern C++ Inference: Retain flexibility

Example 2: `string_view` argument



- Suppose we have a function
  `void f(string_view);`
- Accepting a `string_view` in this way is often recommended in modern C++
- At some point, requirements may change for our program to work for multiple character types and not just `char`

First attempt



- The obvious first attempt is to simply make `f` a function template
- ```
  template<typename charT>
  void f(basic_string_view<charT>);
  ```
- Unfortunately, breaks many valid calls of the original function
- ```
  f("foo"); // Oops! char const * is not a bsv
  f("foo"s); // Oops! string is not a bsv
  ```
- Can we fix? Stay tuned

A brief history of inference in C++



- Before we give the solution to these examples, let’s find out what is in our inferencing toolkit
- Inference has evolved with C++
  - To understand where we are
  - It is helpful to see how we got here

C++98: FTAD



- Inference dates all the way back to the first C++ standard
- With Function Template Argument Deduction (FTAD), you can often call function templates just like regular functions

```
sort(myVec.begin(), myVec.end());
```

- without needing to explicitly specify the template arguments like

```
sort<vector<int>::iterator>
  (myVec.begin(), myVec.end());
```

General principle



- This is a good example of how Function Template Argument Deduction can be considered the first step on the road to Bjarne Stroustrup’s goal of "making generic programming like normal programming”
  - The calling code does not depend on whether `sort` is a function (acting on objects with a virtual `compare()` method) or a function template that compiles in the comparison
- In other words, we have abstracted away the distinction between runtime dispatch and compile-time dispatch

C++11: `auto`



- C++98

```
for(vector<int>::iterator it = myVec.begin();
    it != myVec.end(); it++)
  total += *it;
```

- C++11 `auto`, infers the type of variables from their initializers

```
for(auto it = myVec.begin(); it != myVec.end(); it++)
  total += *it;
```

- Of course, C++11 added complementary features like range-for loops

```
for(auto &x : myVec)
  total += x;
```

C++14: generic lambdas



- C++14 allowed lambdas to infer their arguments

```
for_each(myVec.begin(), myVec.end(),
  [&total](auto x) { total += x; });
```

- where C++11 required

```
for_each(myVec.begin(), myVec.end(),
  [&total](int x) { total += x; });
```

- Not shorter, but less brittle!
- Looser coupling enables changing the type stored in `myVec` without rewriting the loop

C++17: CTAD



- C++17 Class Template Argument Deduction is to object creation what Function Template Argument Deduction is to object use
- C++14 – Locking both sides of an assignment

```
shared_lock<shared_timed_mutex>
  lock_source(r.mut, defer_lock);
lock(l.mut, lock_source);
lock_guard<shared_timed_mutex> llck(l.mut, adopt_lock);
lock_guard<shared_lock<shared_timed_mutex>>
  rlck(lock_source, adopt_lock);
```

- C++17 CTAD

```
shared_lock lock_source(r.mut, defer_lock);
lock(l.mut, lock_source);
lock_guard llck(l.mut, adopt_lock);
lock_guard rlck(lock_source, adopt_lock);
```

C++17: CTAD (continued)



- C++17 `scoped_lock` builds on CTAD to make it even simpler! (It is no coincidence that CTAD and `scoped_lock` were introduced together)

```
shared_lock lock_source(r.mut, defer_lock);
scoped_lock lck(l.mut, lock_source);
```

- Note that this isn't just simpler, it reduces brittleness
  - If `l` and `r` are eventually changed from C++14 `shared_timed_mutex` to C++17 `shared_mutex`, both the C++17 versions remain unchanged

Deduction guides



- CTAD also introduces another feature “deduction guides” suggested by Richard Smith for customizing deduction
- Just deducing from the constructors alone, the following wouldn’t work because we would have to magically deduce we want the value type of the iterator
- `vector v(l.begin(), l.end());`
- But it does work because the deduction has been customized with a deduction guide
- ```
  template<typename Iter>
  vector(Iter b, Iter e)
    -> vector
     <typename iterator_traits<Iter>::value_type>;
  ```
  ```

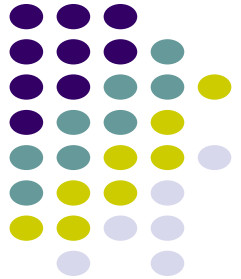
C++20: Concepts



- C++20 has the biggest inference-focused feature yet

Concepts

Unconstrained templates

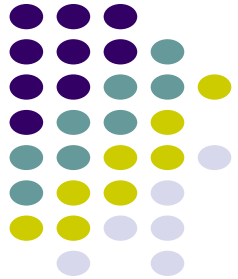


- The basic idea of Concepts is that you can now easily constrain the types that are inferred to those satisfying a condition
- Consider a traditional template

```
template<typename T>
T gcd(T l, T r){
  return r == 0 ? l : gcd(r, l%r);
}
```

- The author of `gcd` probably meant it to be only called on integral types
- But calling it on other types will result in having to figure out compile errors deep in the internals of the algorithm (or simply incorrect answers)
- While this might not be so bad for `gcd`, take a look sometime at the error you get if you make the common mistake of calling `sort` on a `list`!

Old-style constraints

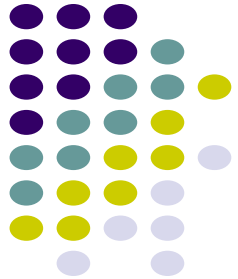


- Recognizing the need to express constraints on template parameters, the standard library provided some (awkward) workarounds

```
template<typename T,
  typename = enable_if_t<is_integral_v<T>>>
T gcd(T l, T r){
  return r == 0 ? l : gcd(r, l%r);
}
```

- In spite of the obvious clumsiness and limitations, `enable_if` is used almost 20,000 times in the ACTCD19 dataset of common C++ code
- This makes clear that there is a demonstrated need for an ability to effectively specify constraints on inferred types

Concept constraints

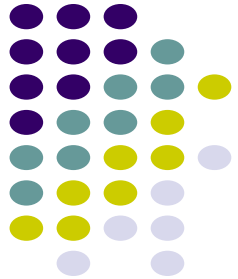


- In C++20 `gcd` can be constrained more naturally as

```
template<Integral T>
T gcd(T l, T r){
  return r == 0 ? l : gcd(r, l%r);
}
```

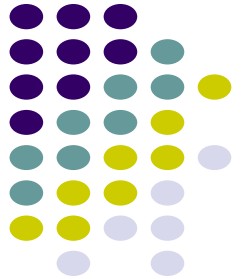
- Just looking at the declaration makes clear that `T` needs to be an integral type
  - and the compiler will reject arguments that are not so
- Unfortunately, Concepts has many features beyond the scope of this talk
  - but there are many other excellent talks at this conference that will do so!

Short form concepts



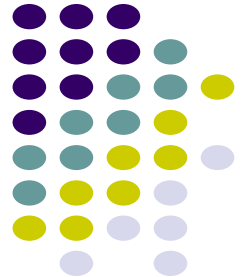
- Concepts also make generic programming more like normal programming by allowing function templates to be declared much more like ordinary functions
- `void f(int x); // function`
- `void f(Integral auto x); // function template`

C++20: `type_identity_t`



- Concepts is the big flashy inference feature in C++20 that every one is talking about
- But there's another C++20 inference feature that you might not even notice that can play a big role in reducing brittleness and tight coupling called `type_identity_t`
- Interestingly, unlike the inference features we have discussed so far, its role is to suppress inference
- Let's take a look

How function templates can introduce brittleness



- Templates can make code hard to work with because they are more tightly coupled to their arguments
- While functions will convert their arguments for you, function templates require exact matches, which can make you tear your hair out

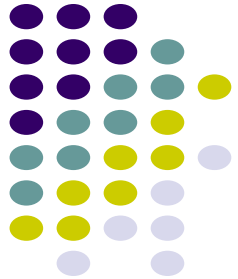
```
void f(unique_ptr<int> ip, int x);
f(make_unique<int>(), 'a'); // OK
```

- Suppose later you want to change `f` to a function template

```
template<Integral T> void f(unique_ptr<T>, T);
f(make_unique<int>(), 'a'); // Oops, is T int or char?
```

- Brittle
  - Changing your mind later and making the function a function template breaks client code
- Even if a function template from the beginning, you probably want code like that to work
  - since the `unique_ptr` argument points authoritatively to what is stored

Fix with `type_identity_t`



- `type_identity_t` can reduce the undesirably finicky tight coupling
- Fix with

```
template<Integral T>
void f(unique_ptr<T>, type_identity_t<T>);
f(make_unique<int>(), 'a'); // OK. f<int>
```

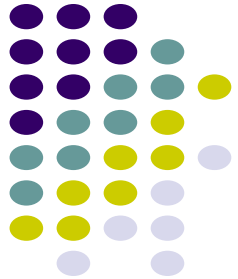
INFERENCE: THE BIG PICTURE

RISING ABOVE THE INDIVIDUAL FEATURES

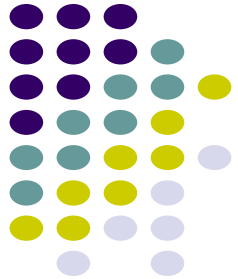
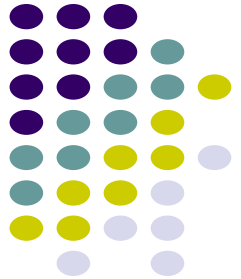
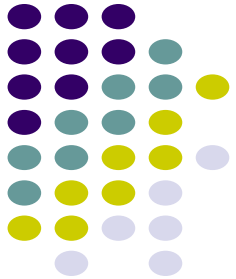
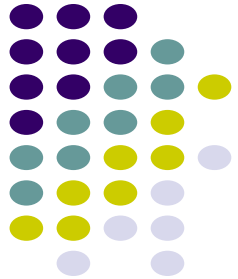
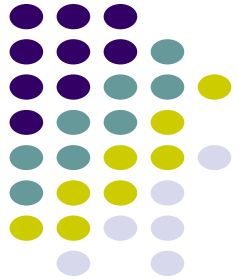
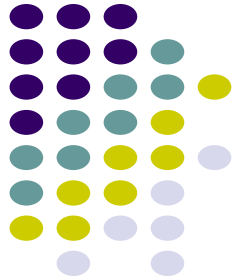
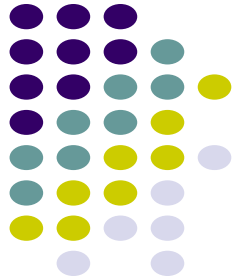
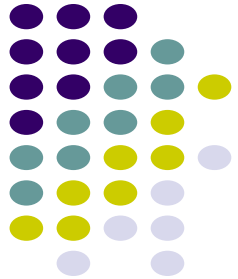
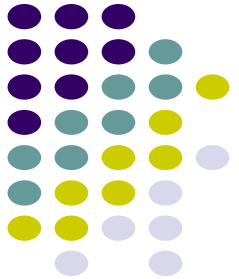
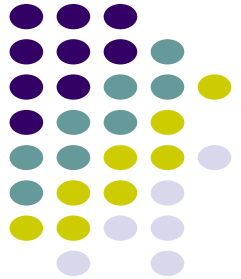
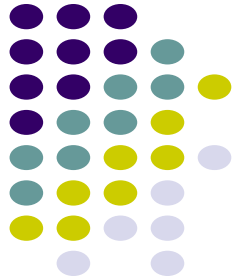
Stop thinking of inference as individual convenience features

- The above features were all added to C++ on their own merits and can be used on their own
- However, if we really want to write robust flexible code that banishes brittleness, we need to stop thinking about inference a feature at a time
- Let’s look at some examples

Warm up: `ranges::to`

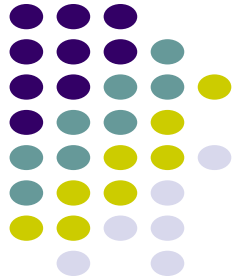


- A nice example of what you can get when FTAD and CTAD (and Concepts) work together to ensure type correct “view materialization”
- ```
  std::list<int> l;
  // c is a vector<int>
  auto c = ranges::to<vector>(l);
  ```

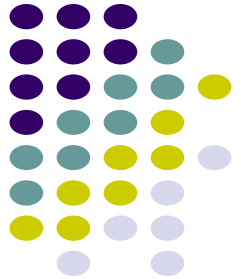
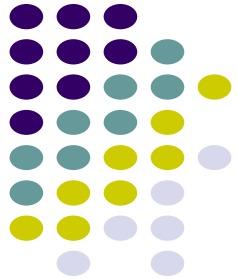
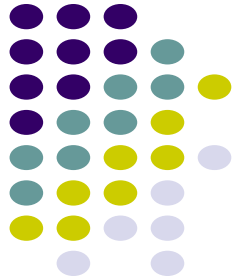
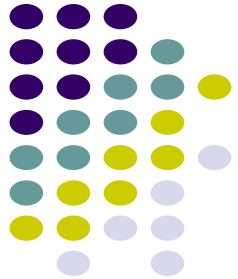
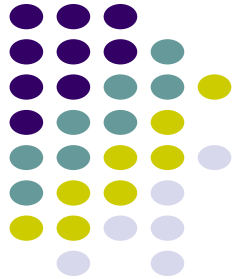
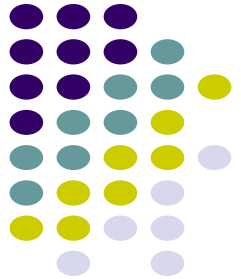
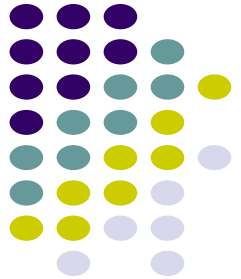
- C++20 status unclear. Available in ranges v3
  ```

Example 1: Turning a base class into a concept



- In the following graphics code, Curve is a base class with virtual methods
-
```
pair<Curve, vector<Point<double>>>
f(Curve &c)
{
  Point<double> top_left{c.x_min(), c.y_max()};
  screen->line(top_left, {c.x_min(), c.y_max()});
  vector<Point<double>>
    cp(c.ctrl_pts.begin(), c.ctrl_pts.end());
  return {c, cp};
};
```
- As graphics programming is often very performance intensive, we may want to look at turning Curve into a concept to improve performance through compile-time
- As we will see ~~later~~**now**, modern C++ makes it much much easier to maintain this flexibility than traditional C++

Curve example: Combining Concepts, CTAD, and FTAD



- Because of concepts constrained auto notation, it will be much more common to have objects with no name for their types
- However, since CTAD and FTAD don’t need explicit type names, you can make function templates look much more like normal programming
- Let’s see

Changing `Curve` into a Concept



- After turning `Curve` into a concept, we reflect this in the signature of `f` by changing argument to `f` to be `Curve auto`
- Note that ugly changes to the return type are also required
  - We'll come back to this

```
auto f(Curve auto &c)
  -> pair<decltype(c),
          vector<typename decltype(c.ctrl_pts)::value_type>>;
```

What about the body



- Since f is a function template, we will need to infer a lot of internal calls and object creations
- This is OK as long as our uses can inference from them

```
auto f(Curve auto &c)
{
  Point top_left{c.x_min(), c.y_max()};
  screen->line(top_left, {c.x_min(), c.y_max()});
  vector
    cp(c.ctrl_pts.begin(), c.ctrl_pts.end());
  return pair{c, cp};
};
```

- Now, changing `Curve` back and forth between a base class and a Concept is as simple as just adding or removing one `auto` from the signature
- Template programming is starting to “look like normal programming”!

What is inference doing here?



- A good way to see what inference is doing is by writing out what `f` would look like without using FTAD or CTAD (except for the argument `c`)

```
auto f(Curve auto &c)
-> pair<decltype(c),
        vector<typename decltype(c.ctrl_pts)::value_type>>
{
  Point<decltype(c.x_min())>
    top_left{c.x_min(), c.y_max()};
  screen->line<decltype(p.x)>(p, {c.x_min(), c.y_max()});
  vector<typename decltype(c.ctrl_pts)::value_type>
    cp(c.ctrl_pts.begin(), c.ctrl_pts.end());
  return {c, cp };
};
```

Inference inside your templates is why you don't need to litter your code with



- `decltype` for expressions without named types
  - This is especially nice for short-form concepts
  - What is the type of the `Curve c`?
- `typename` for expressions with dependent types
  - The above example is just as bad with long-form concepts
- Similar benefits for deduced return types, as is common in code like range pipelines
  - Indeed, Eric Niebler has observed that each stage being able to deduced its template arguments from the return type of the previous is one of the key ingredients in making ranges work

Is template programming always that much like normal programming?



- No! (At least not yet)
- While the above example is nice, and these techniques are often very useful,
- The fact remains that we were lucky…
- Some code intrinsically depends on whether we have a concept or a class
  - `vector<Curve> v; // Curve must be a class`
    `auto fp = &f; // Curve must be a class`
- It is likely that our code still will have dependencies on whether `Curve` is a class or a Concept

Can we do better?



- If we want to retain maximum flexibility as to whether a class might later become a concept, wrap it with a concept
- See https://github.com/mspertus/share/blob/master/include/ClassWrap.h for wrappers that make `f` completely agnostic as to whether `Curve` is a class or a concept
  - Example godbolts at http://bit.ly/ClassWrap_2kTMK4z and http://bit.ly/ClassTemplateWrap_2m25YoB
  - You don't even need to add/remove `auto` anymore
  - You are prevented from inadvertently putting in code that hardwires assumptions whether it is a class or concept

`string_view`: Combining FTAD, CTAD, and Concepts



- Let's return to our `string_view` example
  `void f(string_view);`
- At some point, it is certainly reasonable that we may get the requirement that our program works for multiple character types and not just char
- Let's see how this can be accomplished by using FTAD, CTAD, and Concepts together

`string_view` example



- The obvious first attempt is to simply make `f` a function template
- ```
  template<typename charT>
  void f(basic_string_view<charT>);
  ```
- Unfortunately, `f` no longer works in many of the cases where it did before
- ```
  f("foo"); // Oops! char const * is not a bsv
  f("foo"s); // Oops! string is not a bsv
  ```

Workaround with overloads



- Sometimes people try to workaround this by providing many overloads
-
```
template<typename charT, class Tr>
void f(basic_string_view<charT, Tr>);

template<typename charT>
void f(charT const *s)
  { f(basic_string_view<charT>(s); }

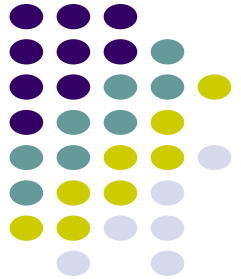
template<typename charT>
void f(charT const *s, unsigned len)
  { f(basic_string_view<charT>(s, len); }

template<typename charT, class Tr, class Alloc>
void f(basic_string<charT, Tr, Alloc> s)
  { f(basic_string_view<charT, Tr>(s); }
```

Not only does this kind of defeat the purpose of `string_view`



- Indeed, `std::quoted` from the standard library takes this approach
- This is tedious and error-prone
- And brittle because if a new constructor is added to `string_view`, `f` becomes silently out of date
- Virtually impossible to keep in sync
- Indeed, LEWG has passed P1391, making `string_view` constructible from a range but the proposed wording neglects to update `std::quoted`

CTAD to the rescue

- It seems like what we need is a way to deduce which `basic_string_view` we want and then construct it
- But this is exactly what CTAD does!
-
```
// http://bit.ly/CPPCon19_FTAD_2ko1lVA
template<typename T>
void f_impl(basic_string_view<T>);

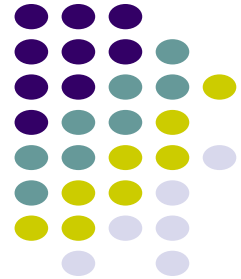
template<typename T> void f(T &&t)
{
  f_impl(basic_string_view(forward<T>(t)));
}

f("hello"); // OK. T is char
f(basic_string_view(L"world")); // OK. T is wchar_t
```

Are we done?



- This has certainly helped us create a function template that accepts a `basic_string_view`, but it has messed up overloading in the process
- Suppose we also have

```
void f(Integral auto);
f(5); // Oops f(T &&) matches too
```

Concepts to the rescue



- `// http://bit.ly/CPPCon19_FTAD_Concepts_2kny8Kn`

```
template<typename T>
concept DeducesBasicStringView
  =  requires(T x) {
  basic_string_view(x);
};
```

-
```
void
f(DeducesBasicStringView auto &&t)
{ f_impl(basic_string_view(t)); }
```

- This is what we want!

The Good



- Brittleness dispelled: We were able to change our function to work with any character type
- This required essential use of Function Template Argument Deduction, Class Template Argument Deduction, and Concepts
- With C++20, even more complex cases like

```
template<class T> void f(X<T*>);
```

can be handled (by leveraging CTAD for alias templates) as in https://github.com/mspertus/share/blob/master/examples/FTADExamples.cpp

The bad



- Not a perfectly transparent change. For example, we can't take `&f`
- Still have to say
  ```
  f(basic_string_view{"foobar", 3});
  ```
- We still need an overload for `basic_string` to handle
  ```
  f("foo"s);
  ```
  - The problem is that `basic_string_view` lost its `basic_string` constructor between LFTS and C++17
  - Need a deduction guide `basic_string_view->basic_string`
  - Or better yet, CTAD should deduce from `operator foo()` typecasts, which would have handled this transparently (this was my bad)

The Ugly



- Yeah, it’s a bit of a kludge
- The problem is that we’ve built up inference as a series of features that are not seamless
- Maybe one day they will be
  - There are good proposals addressing all of the above
- But in the meantime, that is why we learn useful idioms, just like we used to do with `enable_if_t`

Combining CTAD with `type_identity_t`



- pmr::vector is defined as

```
namespace pmr {
template<typename T>
using vector
 = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

- Unfortunately, inference only sort of works (the following uses C++20 CTAD for type aliases)
- The following work

```
vector v = {1, 2, 3}; // OK
pmr::vector v = {1, 2, 3}; // OK
pmr::vector v({1, 2, 3}, pmr::allocator(mr));
```

- But this does not, because pmr::memory_resource is not a pmr::allocator

```
pmr::vector v({1, 2, 3} mr); // Oops!
```

A better way



- Using `type_identity_t` in your alias templates lets you control what is inferenced
- ```
  namespace pmr {
  template<typename T>
  using vector
   = std::vector<T,
  type_identity_t<pmr::polymorphic_allocator<T>>>;
  }
  ```
- Conveniently, this results in no API breakage to speak of because type_identity_t<T> is just another name for T!
  ```

A great example of transparently turning a class into a class template in an existing codebase



- Communication from Mark Zeren at VMWare
- https://godbolt.org/z/xymaTH
- Matt Godbolt always says not to look at instruction count to tell what code runs faster
- But after I showed him this one, he said it was OK in this case!
- Could one generate similar code without this?
  - Yes, but it is usually done with a macro ☹
  - This was a transparent drop-in of a class template for a class in a large production codebase
  - Again illustrating how modern inference can be used to combat brittleness and enhance codebases



BEST PRACTICES

AACA: Combining concepts and auto

- The more you repeat the same information, the more difficult it becomes to keep your code in sync

```
unsigned x = f(); /* unsigned f() */
```

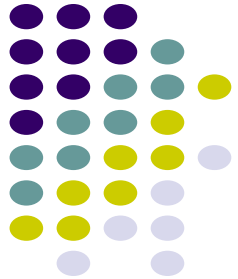
- Suppose later you discover that you really needed an `unsigned long` rather than an `unsigned` to scale up your app
- So you change the return type of to `unsigned long` but you inadvertently fail to update the type of `x`

```
unsigned x = f(); /* Oops! Truncation */
```

- For reasons like these, a style known as *Almost Always Auto* has become popular

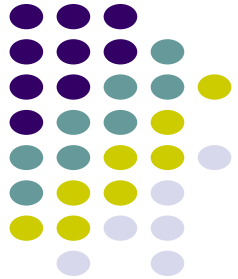
```
auto x = f(); /* OK */
```

Almost always auto



- As the above example shows, deducing an object to have the same type as its initializer can make code less brittle
- IDEs are doing their part to support such coding styles
  - For example, Visual Studio will show the deduced type for `c` if you mouse over it
  - Although it's still a work in progress as I am often annoyed because I can't press F12 to go to the definition of the deduced type

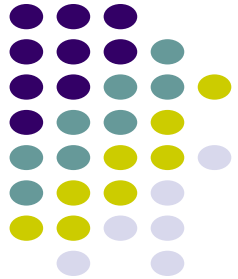
So what's the problem?



- Examples like the above are popular, but they can also increase program brittleness
- Suppose changing the return type of `f` means that the call site now needs to change its logic
  - Just deducing everything doesn't mean the code will magically work
  - For example, the code may assume that `x` is an unsigned type
- This is not unlike the problem of unconstrained inference that we used to motivate Concepts above
- And indeed, Concepts provide a better solution with Constrained Auto
- The client code can add constraints to describe what types it can process

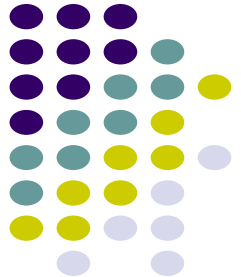
```
unsigned_integral auto x = f(); // Safer
```

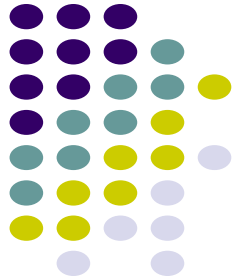
Almost Always Constrained Auto



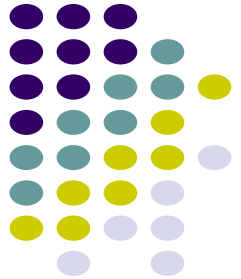
- Almost Always Constrained Auto is superior in every way to Almost Always Auto
- So wherever you would declare a variable with `auto` today, if you have any suitable concepts, be sure to constrain your declaration as in the example above
  - If there is no such concept, strongly consider defining one
- This is (essentially) T.12 in the core guidelines

Best Practice: Extending `make_unique`



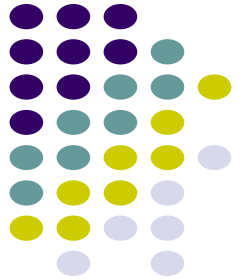
- While CTAD works for static or automatic duration objects, it doesn't work with heap objects
- ```
  // Static duration works
  optional o(5); // OK. optional<int>

  // Try to create dynamic object.
  auto o1 = new optional(5);
  // Already violates both core guidelines R.3 and R.11
  auto o2 = unique_ptr(o1);
  // Ill-formed. Can't deduce unique_ptr from raw ptr
  ```
- Bad inconsistency: Whether you deduce objects should have nothing to do with whether you create it on the heap
  ```

Extending `make_unique`



- An additional overload for `make_unique`, `make_shared`, etc. is what's needed here
- ```
  template
  <template <typename ...U> typename T, typename ...A>
  auto make_unique(A&& ...a){
    return unique_ptr
     <decltype(T(forward<A>(a)...))>
    (new T(forward<A>(a)...));
  }
  ```
- Now things work as expected
  ```
  // http://bit.ly/godbolt_p1069_2m0O8Ti
  auto o2 = make_unique<optional>(5);
  ```
- P1069 targeting C++23. Until then, available on Github at https://github.com/mspertus/share/blob/master/include/make_unique.h
  ```

Best practice: Constraining classes



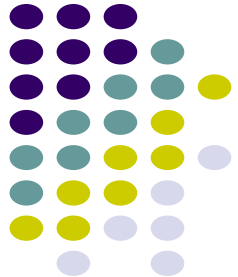
- With SFINAE, people are in the habit of constraining function templates and method templates, but constraining classes is not so common
- This kind of “withholding information” from the compiler has consequences
- Unsurprisingly, properly expressing your class constraints will improve inference
- For example, many of the deduction guides in the standard library would not be needed if the container classes were fully concept constrained
- http://qr.w69b.com/g/qE1JDD0Sk

More CTAD Best Practices



- https://github.com/CppCon/CppCon2017/blob/master/Posters/Best%20Practices%20for%20Constructor%20Template%20Argument%20Deduction/Best%20Practices%20for%20Constructor%20Template%20Argument%20Deduction%20-%20Mike%20Spertus%20-%20CppCon%202017.pdf

Conclusions: C++20 Inference



- Inference is great for getting rid of repetitive boilerplate, but it is much more than that
- Reduces brittleness and tight coupling by enabling generic programming to be more like normal programming
- Should be looked at as a coherent whole rather than a bunch of individual features
  - FTAD and CTAD enable inferenced use and creation of objects
  - Such inference should be constrained with Concepts
  - Supporting features like `type_identity_t` can make inference more friendly by further loosening coupling when appropriate
- Improved support for inference in each successive C++ standard keeps making generic programming more like normal programming
  - A trend we hope will continue