Modern Template Techniques

Jon Kalb
Meeting C++
Berlin
2019-11-16

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Template Challenge

Let's write a function template from scratch.

Don't worry it's the simplest function in the world: `add()`—it adds two things:

```
template <class T>
T add(T a, T b)
{ return a + b; }
```

T is an unknown type so it might be expensive to copy, so:

```
template <class T>
T add(T const& a, T const& b)
{ return a + b; }
```

Easy-peasy, right? The simplest function in the world.

Wait, but if we want to add two things of different types?

```
template <class T, class U>
T add(T const& a, U const& b)
{ return a + b; }
```

Template Challenge

What is wrong with this?

```
template <class T, class U>
T add(T const& a, U const& b)
{ return a + b; }
```

Template Challenge

What is wrong with this?

```
template <class T, class U>
T add(T const& a, U const& b)
{ return a + b; }
```

What should the return type be?

- We can't assume it is the type of the first parameter.
- We can't even assume it is the type of either parameter.
  - Adding a `char` and a `short` results in an `int`.

The return type should be *whatever you get when you add a T and a U*.

How do we say that?

```
template <class T, class U>
decltype(T + U) add(T const& a, U const& b)
{ return a + b; }
```

Template Challenge

What's wrong with this?

```
template <class T, class U>
decltype(T + U) add(T const& a, U const& b)
{ return a + b; }
```

It doesn't compile!

Why not?

`decltype` works on expressions. You can't add two types.

How can we fix this?

```
template <class T, class U>
decltype(a + b) add(T const& a, U const& b)
{ return a + b; }
```

Template Challenge

What's wrong with this?

```
template <class T, class U>
decltype(a + b) add(T const& a, U const& b)
{ return a + b; }
```

It doesn't compile!

Why not?

`a` and `b` in `decltype(a + b)` are not defined.

How can we fix this?

```
template <class T, class U>
auto add(T const& a, U const& b) -> decltype(a + b)
{ return a + b; }
```

Template Challenge

```
// C++11
template <class T, class U>
auto add(T const& a, U const& b) -> decltype(a + b)
{ return a + b; }
```

This is the simplest template in the world.

But it couldn't be written in Classic C++ because we need *decltype* and *trailing return type function declarations*.

Both of these were introduced in C++11 and they are important tools in even basic template code.

Template Challenge

```
// C++11
template <class T, class U>
auto add(T const& a, U const& b) -> decltype(a + b)
{ return a + b; }
```

C++14 made it even easier to create templates by allowing us to use ***type deduction*** for return types.

Template Challenge

```
// C++14
template <class T, class U>
auto add(T const& a, U const& b)
{ return a + b; }
```

C++14 made it even easier to create templates by allowing us to use ***type deduction*** for return types.

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Static Polymorphism

- *Polymorphism*:
  - a common interface
    - defined by a base class
    - implemented by derived class
- *Dynamic polymorphism*: relies on tools from Object-Oriented Programming.
  - Using a base class pointer (or reference) to a derived class object:
    - We don't know actual type at compile time.
    - Virtual functions
      - Indirect dispatching at runtime.
- *Static polymorphism*: relies on Curiously Recurring Template Pattern
    - We know the actual type at compile time
    - No need for virtual functions or runtime indirection
    - Allows us to inject behavior into a class without v-table

CRTP

- Curiously Recurring Template Pattern:

```
struct derived: base<derived>
{
  ~~~
}
```

- *Does that even compile?*

CRTP

- Challenge:
  - We want to create a base class with an interface that will be implemented by derived classes, but *without virtual functions*.
  - We can rely on knowing the type of the derived class at compile time, but we only want to use the inherited interface.
  - We want this to type-safe.

CRTP

```
template <class Derived>
struct base
{
  void interface()
  {
    // verify pre-conditions, etc
    static_cast<Derived*>(this)->implementation();
    // verify post-conditions, etc
  }
};

struct derived: base<derived>
{
  void implementation() { /* */ }
};

derived d;
d.interface(); // Uses the base class interface to get derived behavior.
```

CRTP

- The traditional first example from Steve Dewhurst:

```
template <class T>
struct counter
{
  counter() {++ctr_;}
  counter(counter const&) {++ctr_;}
  ~counter() {--ctr_;}
  static long get_count() {return ctr_;}
  private:
  inline static long ctr_;                   // inline variables from C++17
};

struct my_string: counter<my_string> {~~~};
my_string a, b{"content"};
my_string c{a};

std::cout "count: " << my_string::get_count() << "\n";

count: 3
```

CRTP

- Simple example:

```
template <class Derived>
struct cloneable
{
  Derived* clone() const
  { return new Derived{static_cast<Derived const&>(*this)}; }
};

struct bar final: cloneable <bar>
{
  bar(int id): id{id} {}
  int id;
};

bar b{42};
bar* my_clone{b.clone()};
std::cout << "id: " << my_clone->id << "\n";

id: 42
```

CRTP

- More interesting example (thanks to Barton, Nackman, and Dewhurst):

```
template <class T>
struct eq
{
  friend bool operator==(T const&a, T const&b) {return a.compare(b) == 0;}
  friend bool operator!=(T const&a, T const&b) {return a.compare(b) != 0;}
};
```

- Where compare() is defined in the derived class and returns a negative value for less, zero for equals, and a positive value for greater.

```
template <class T>
struct rel
{
  friend bool operator<(T const&a, T const&b) {return a.compare(b) < 0;}
  friend bool operator<=(T const&a, T const&b) {return a.compare(b) <= 0;}
  friend bool operator>(T const&a, T const&b) {return a.compare(b) > 0;}
  friend bool operator>=(T const&a, T const&b) {return a.compare(b) >= 0;}
};
```

CRTP

```
template<class T> struct my_complex: eq<my_complex>
{
  T real;
  T imaginary;

  bool compare(my_complex const&rhs) const
    {return (real == rhs.real) and (imaginary == rhs.imaginary);}
  ~~~
};

struct my_string: eq<my_string>, rel<my_string>
{
  bool compare(my_string const&rhs) const {return std::strcmp(s, rhs.s);}
  ~~~

  private:
  char* s;
};
```

CRTP

- Real world example:
  - In our application, Widgets are always in shared pointers.

```
std::vector<std::shared_ptr<Widget>>       // data structure for
  processedWidgets;                        // processed Widgets

struct Widget {
  …
  void process() {                         // Widget-processing function
    …                                      // process the Widget
    processedWidgets.emplace_back(this);   // uh oh…
  }
};
```

- This is a problem waiting to happen. `this` is a raw pointer, so:
  - Call to `emplace_back` creates a control block for `*this`.
- But there is already at least one `std::shared_ptrs` pointing to `*this`.
  - So, we have Undefined Behavior.

CRTP

- Real world example: Solution
  - `std::enable_shared_from_this` (a CRTP type)

```
std::vector<std::shared_ptr<Widget>>       // data structure for
  processedWidgets;                        // processed Widgets

struct Widget: std::enable_shared_from_this<Widget> {
  …
  void process() {                         // Widget-processing function
    …                                      // process the Widget
    processedWidgets.emplace_back(shared_from_this());
  }
};
```

- Inherit `std::enable_shared_from_this` to safely convert `this` to `shared_ptr`.
  - Call to `shared_from_this` instead of using `this`.
- But how does this work?

CRTP

- Real world example: Solution
  - `std::enable_shared_from_this` (a CRTP type)

```
template <class T> struct enable_shared_from_this
{
  shared_ptr<T> shared_from_this()
  {
    shared_ptr<T> p{weak_this_}; return p;
  }
  shared_ptr<T const> shared_from_this() const
  {
    shared_ptr<T const> p{weak_this_}; return p;
  }
private:
  mutable weak_ptr<T> weak_this_;
};
```

- Because the base class is template on the type of the derived class, it can have a data member that is a weak_ptr to that class.

CRTP

- Consider:
  - There is a typo in this code

```
struct derived1: CRTP_base<derived1>
{
  void implementation();
  ~~~
};
struct derived2: CRTP_base<derived1>
{
  void implementation();
  ~~~
};
```

  - What is the typo?

CRTP

- Consider:
  - There is a typo in this code

```
struct derived1: CRTP_base<derived1>
{
  void implementation();
  ~~~
};
struct derived2: CRTP_base<derived1> // Wrong base class template parameter
{
  void implementation();
  ~~~
};
```

  - What is the typo?
- How do we prevent this?

CRTP

- Preventing bad inheritance:

```
template <class Derived> struct CRTP_base
{
  ~~~
  private:
  CRTP_base() = default;
  friend Derived;
};
```

- How/Why does this work?
  - With private constructor, a class can only instantiated by a member or friend.
  - In this case, CRTP_base can only instantiated by the class that is its template parameter so `struct derived2: CRTP_base<derived1>` can be declared but not instantiated.

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Type Traits

Type traits are compile-time query-able type characteristics that we can use to emit different code at compile-time so there is no run time overhead.

We'll just explore some simple characteristics.

Can we determine if a template type is an int?

```
template <class T>
struct is_int
{ static bool const value{false}; };

template<>
struct is_int<int>
{ static bool const value{true}; };

std::cout << "float : " << is_int<float>::value << " ";
std::cout << "int : " << is_int<int>::value << " " ;
```

```
float: 0 int: 1
```

Type Traits

The convention is that the result of a type trait is named "*type_trait*::value."

Starting in C++17 we can have templated inline constexpr variables, so the convention is to declare an "_v" variable for each type trait:

```
template <class T>
constexpr bool is_int_v{is_int<T>::value};
```

Type Traits

Can we determine if two types are the same?

```
template <class T, class U>
struct is_same { static bool const value{false}; };

template<class T>
struct is_same <T, T> { static bool const value{true}; };

template <class T, class U>
constexpr bool is_same_v{is_same<T, U>::value};

std::cout << "same : " << is_same_v<float, int> << "\n";
std::cout << "same : " << is_same_v<int, int> << "\n";
```

```
same : 0
same : 1
```

Type Functions

- These examples have been type traits– characteristics that returned a compile-time constant Boolean value.
- Let's try some *type functions*. These are compile-time expressions that take types and return types.
- The convention is that the result of a type function is a type alias named "*type_function*::type."

In Modern C++ we usually create a template type alias with the name *type_function*_t.

Type Functions

Let’s create a function that removes const.

```
template <class T>
struct remove_const
{ using type = T; };

template<class T>
struct remove_const <T const>
{ using type = T; };

template <class T>
using remove_const_t = typename remove_const<T>::type;

std::cout << is_same_v<int, remove_const_t<int const>> << "\n";
std::cout << is_same_v<int const, remove_const_t<int const>> << "\n";

1
0
```

<type_traits>

Primary type categories:

`is_void, is_null_pointer, is_integral, is_floating_point, is_array, is_enum, is_union, is_class, is_function, is_pointer, is_lvalue_reference, is_rvalue_reference, is_member_object_pointer, is_member_function_pointer`

Composite type categories:

`is_fundamental, is_arithmetic, is_scaler, is_object, is_compound, is_reference, is_member_pointer`

Type properties:

`is_const, is_volatile, is_trivial, is_trivially_copyable, is_standard_layout, is_pod, is_literal_type, is_empty, is_polymorphic, is_final, is_abstract, is_signed, is_unsigned`

Supported operations:

Note: each * is replaced by, “”, “`is_trivially_`”, and “`is_nothrow`”.

`is_*constructable, is_*default_constructable, is_*copy_constructable, is_*move_constructable, is_*assignable, is_*copy_assignable, is_*move_assignable, is_*destructable, has_virtual_destructor`

Property queries and type relationship:

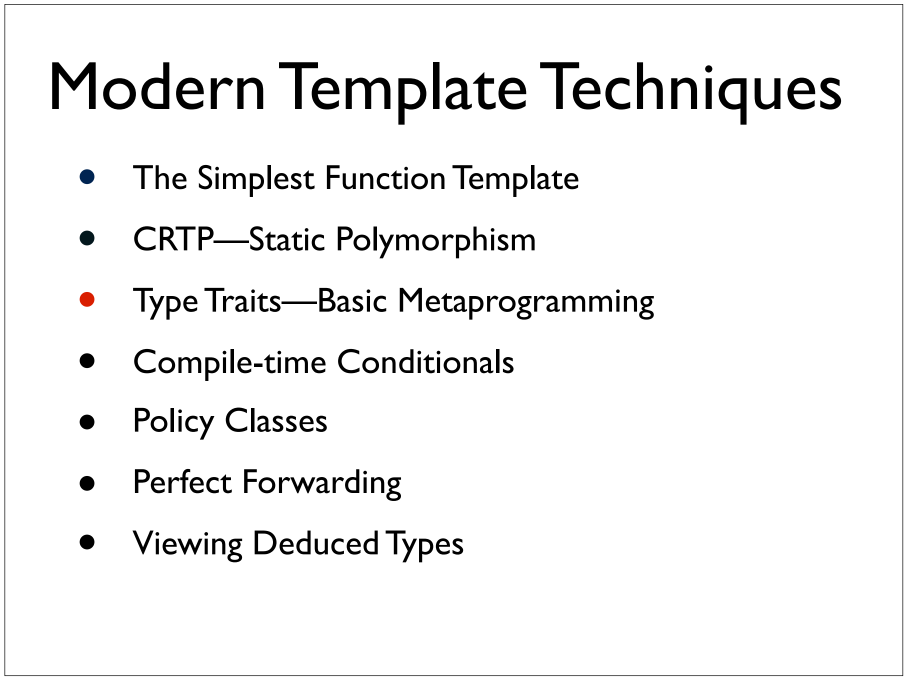
`alignment_of, rank, extent, is_same, is_base_of, is_convertable`

Qualifiers, References, Pointers, Sign modifiers, and Arrays:

`remove_cv, remove_const, remove_volatile, add_cv, add_const, add_volatile, remove_reference, add_lvalue_reference, add_rvalue_reference, remove_pointer, add_pointer, make_signed, make_unsigned, remove_extent, remove_all_extents`

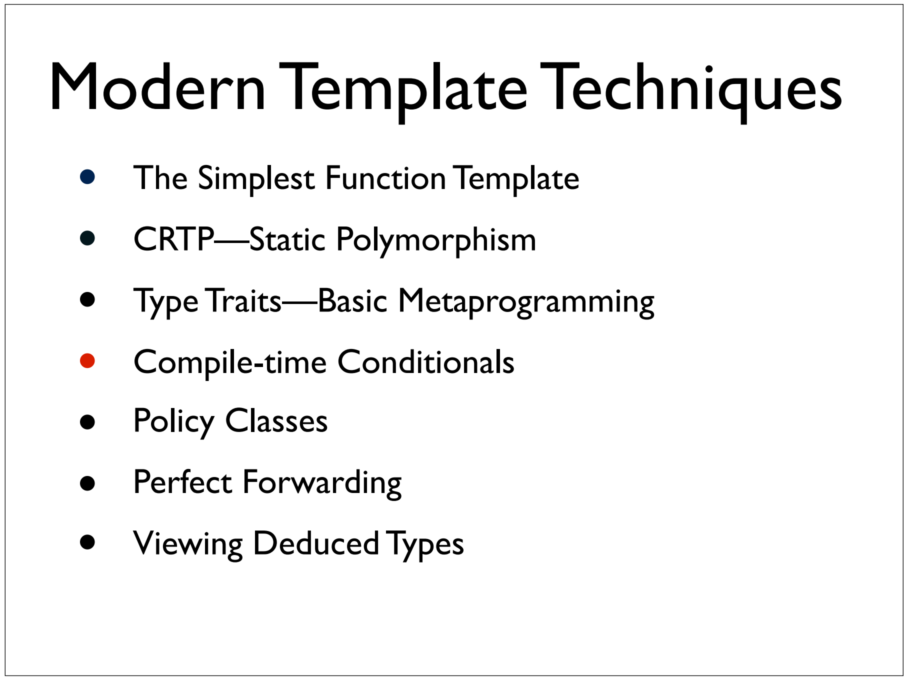
Misc:

`aligned_storage, aligned_union, decay, enable_if, conditional, common_type, underlying_type, result_of, void_t`



Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types



Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Compile-time Conditionals

- Tag Dispatch
- Compile-time Conditional Overloading
- Constexpr Conditional

Tag Dispatch

The standard defines a function called *std::distance* that returns the number of times that a given iterator must be incremented to have the same value as different iterator.

(Undefined if the two iterators do not reference the same range.)

What is the complexity of this function?

Tag Dispatch

How can we implement *distance* so that it is correct for any type of iterator, but is constant time for *random access* iterators?

The two implementations are easy:

```
template <class Iterator>
auto distance(Iterator first, Iterator last)
{
  typename std::iterator_traits<Iterator>::difference_type result{0};
  while (first != last) ++first, ++result;
  return result;
}

template <class RAIter>
auto distance(RAIter first, RAIter last)
{
  return last - first ;
}
```

Tag Dispatch

```
template <class Iterator>
auto distance(Iterator first, Iterator last)
{
  typename std::iterator_traits<Iterator>::difference_type result{0};
  while (first != last) ++first, ++result;
  return result;
}

template <class RAIter>
auto distance(RAIter first, RAIter last)
{
  return last - first ;
}
```

What's wrong this?

redefinition of distance

Tag Dispatch

```
template <class Iterator>
auto distance(Iterator first, Iterator last)
{
  typename std::iterator_traits<Iterator>::difference_type result{0};
  while (first != last) ++first, ++result;
  return result;
}

template <class RAIter>
auto distance(RAIter first, RAIter last)
{
  return last - first;
}
```

What's wrong this?

redefinition of distance

To solve this problem using *tag dispatching*, we add a *tag* parameter to each overload and then add a function that calls the correct overload.

Tag Dispatch

```
template <class Iterator>
auto distance(Iterator first, Iterator last, std::input_iterator_tag)
{
  typename std::iterator_traits<Iterator>::difference_type result{0};
  while (first != last) ++first, ++result;
  return result;
}

template <class RAIter>
auto distance(RAIter first, RAIter last, std::random_access_iterator_tag)
{   return last - first;  }
```

We call these additional parameters "tags" because we will don't use them for passing values.

Note: no parameter name.

We use them only so the compiler calls the right version.

Tag Dispatch

```
template <class Iterator>
auto distance(Iterator first, Iterator last, std::input_iterator_tag);

template <class RAIter>
auto distance(RAIter first, RAIter last, std::random_access_iterator_tag);
```

Now we need to define the dispatching function.

```
template <class Iterator>
auto distance(Iterator first, Iterator last)
{
  return distance(first, last,
            typename std::iterator_traits<Iterator>::iterator_category{});
}
```

Dispatched at compile-time. No runtime cost.

We are leveraging the *iterator_traits/iterator_category* infrastructure provided by the standard.

We can make up our own types as needed.

Compile-time Conditionals

- Tag Dispatch
- Compile-time Conditional Overloading
- Constexpr Conditional

Compile-time Conditionals

- Tag Dispatch
- Compile-time Conditional Overloading
- Constexpr Conditional

Compile-time Conditional Overloading

We want to create a function for copying an array.

```
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N])
{ std::copy(source, std::end(source), dest); }
```

What if T is an int?

It would likely be faster to call memcpy.

(A good std::copy implementation will detect this situation and do that for us.) But what if we want to do it ourselves? We cannot partially specialize, but we can overload.

```
template <std::size_t N>
void copy_array(int (&source)[N], int (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(int)); }
```

Compile-time Conditional Overloading

Yeah!

```
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N])
{ std::copy(source, std::end(source), dest); }

template <std::size_t N>
void copy_array(int const (&source)[N], int (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(int)); }
```

What if T is a float?

We could be doing a lot of overloads. What we want is this:

```
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(T)); }
```

What's wrong with this?

redefinition of copy_array

Compile-time Conditional Overloading

What we'd like to be able to have both definitions but only use the one that works for the type we using.

```
// use if not memcpy safe
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N])
{ std::copy(source, std::end(source), dest); }

// use if memcpy safe
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(T)); }
```

We could *tag dispatch*, but instead we'll *conditionally overload* using *enable_if*.

Compile-time Conditional Overloading

The standard calls "memcpy safe" *trivially copyable*.

```
// use if not memcpy safe
template <class T, std::size_t N>
std::enable_if_t<not std::is_trivially_copyable_v<T>, void>
copy_array(T const (&source)[N], T (&dest)[N])
{ std::copy(source, std::end(source), dest); }

// use if memcpy safe
template <class T, std::size_t N>
std::enable_if_t<std::is_trivially_copyable_v<T>, void>
copy_array(T const (&source)[N], T (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(T)); }
```

enable_if_t takes two parameters.

First is a Boolean compile-time constant expression.

The second is the type that is used if the expression is true.

Compile-time Conditional Overloading

The second defaults to void, so here, we can use the that.

```
// use if not memcpy safe
template <class T, std::size_t N>
std::enable_if_t<not std::is_trivially_copyable_v<T>>
copy_array(T const (&source)[N], T (&dest)[N])
{ std::copy(source, std::end(source), dest); }

// use if memcpy safe
template <class T, std::size_t N>
std::enable_if_t<std::is_trivially_copyable_v<T>>
copy_array(T const (&source)[N], T (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(T)); }
```

A better name for *enable_if* would be *ignore_unless*.

If a template function contains an *std::enable_if* expression then the function is considered for overloading only if the condition evaluates to true.

Compile-time Conditional Overloading

```
// use if not memcpy safe
template <class T, std::size_t N>
std::enable_if_t<not std::is_trivially_copyable_v<T>>
copy_array(T const (&source)[N], T (&dest)[N])
{ std::copy(source, std::end(source), dest); }

// use if memcpy safe
template <class T, std::size_t N>
std::enable_if_t<std::is_trivially_copyable_v<T>>
copy_array(T const (&source)[N], T (&dest)[N])
{ std::memcpy(dest, source, N * sizeof(T)); }
```

SFINAE is *Substitution Failure Is Not An Error*. This mean that instead of failing to compile, the template is just ignored if the compiler can't find a legal substitution.

If the expression evaluates to false, the function is "disabled" by *SFINAE* that is, it is removed from the *overload set* and thus from consideration when copy_array is called.

The set of candidate functions for any particular function call.

Compile-time Conditional Overloading

How would we implement *enable_if* ?

The *true* case is easy:

```
// primary template
template <bool B, class T = void>
struct enable_if;              // forward declaration is an incomplete type

// partial specialization
template <class T = void>
struct enable_if<true, T> {using type = T;}        // true case
```

How do we implement the *false* case?

We don't!

Compile-time Conditional Overloading

Style note:

The *enable_if* type can be a return type or the type of any part of the function signature.

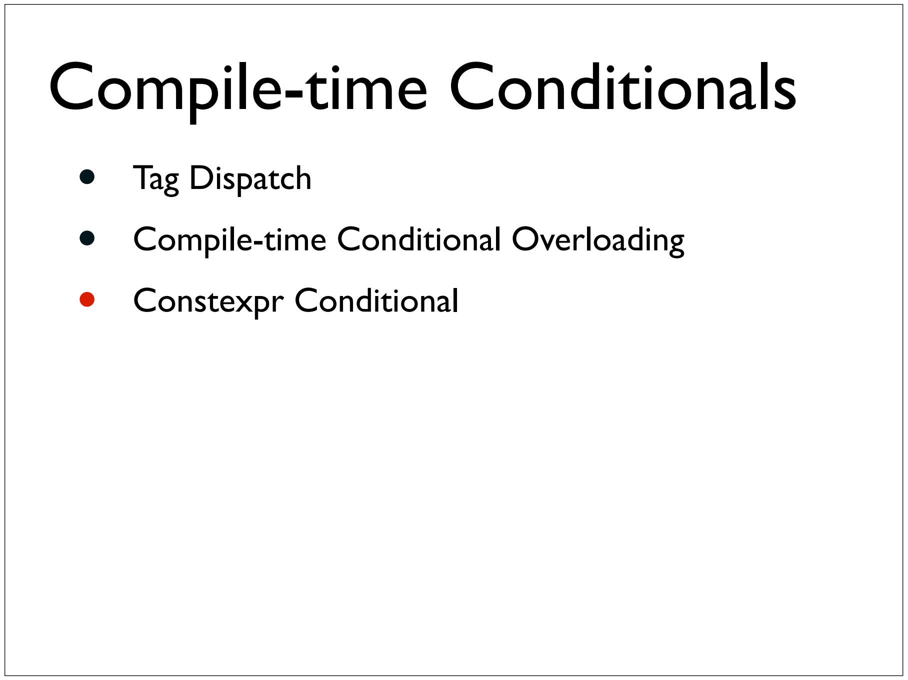
I used the return type, which may be confusing to read.

The popular approach is to add a parameter *with a default value*, whose type is the enable_if type.

```
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N],
std::enable_if_t<std::is_trivially_copyable_v<T>, int> = 0)
{
    std::memcpy(dest, source, N * sizeof(T));
}
```

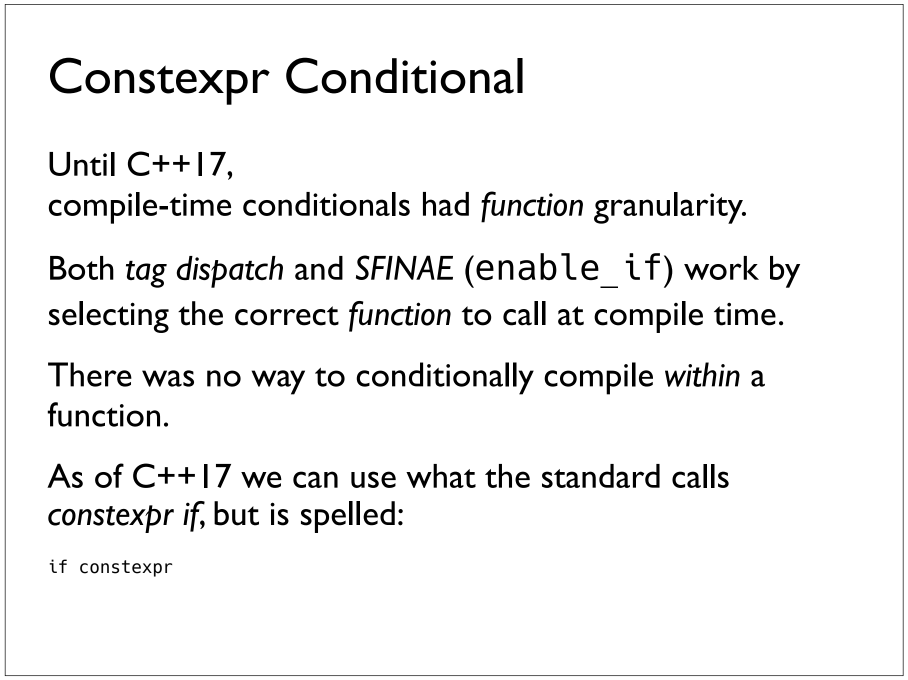
Compile-time Conditionals

- Tag Dispatch
- Compile-time Conditional Overloading
- Constexpr Conditional



Compile-time Conditionals

- Tag Dispatch
- Compile-time Conditional Overloading
- Constexpr Conditional



Constexpr Conditional

Until C++17,
compile-time conditionals had *function* granularity.

Both *tag dispatch* and *SFINAE* (`enable_if`) work by selecting the correct *function* to call at compile time.

There was no way to conditionally compile *within* a function.

As of C++17 we can use what the standard calls *constexpr if*, but is spelled:

```
if constexpr
```

Constexpr Conditional

```
template <class T, std::size_t N>
void copy_array(T const (&source)[N], T (&dest)[N])
{
    if constexpr (std::is_trivially_copyable_v<T>)
      {
      std::memcpy(dest, source, N * sizeof(T));
      }
    else
      {
      std::copy(source, std::end(source), dest);
      }
}
```

Compile-time constant Boolean expression.

For branches not taken, statements are *discarded*.

Constexpr Conditional

Compile-time conditionals inside functions:

- Fewer functions/less duplication
- Easier to read/comprehend
- Easier to maintain

Compile-time Conditionals

- Tag Dispatch
- Compile-time Conditional Overloading
- Constexpr Conditional

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Policy Classes

Designing is a process of making decisions.

When designing a library type, sometimes the correct decision is obvious.

But sometimes, we wish we could ask the user, *What would you want here?*

Policy classes allow us to architect our library types so that some decisions (policies) are provide by the user rather than the library author.

(The library author does provide the customization points for the policy class to customize.)

Policy Classes

Imagine we are creating a type that supports the indexing operator.

We must answer the question, what, if any, checking do we want to do on the provided index and what do we want to do if the checking fails?

Imagine our class template, "MyContainer," is designed to have a CheckingPolicy class as a template parameter. Our template derives from the policy class type.

```
template <class T, class CheckingPolicy>
struct MyContainer: private CheckingPolicy
{
  ~~~
};
```

Policy Classes

```
template <class T, class CheckingPolicy>
struct MyContainer: private CheckingPolicy
{
    ~~~
};
```

To compile, the checking policy class must have a member function that looks like this:

```
template <class U>
void CheckBounds(U const& lower, U const& upper, U const& index);
```

The library can provide some classes that implement the obvious policies, but the library user can create custom policies as well.

Policy Classes

Possible implementations include:

```
template <class U>
void CheckBounds(U const& lower, U const& upper, U const& index) noexcept
{} // Do nothing.

template <class U>
void CheckBounds(U const& lower, U const& upper, U const& index) noexcept
{assert((!(index < lower)) and (index < upper));} // assert

template <class U>
void CheckBounds(U const& lower, U const& upper, U const& index)
{if ((index < lower) or !(index < upper)) throw std::range_error{};} // throw
```

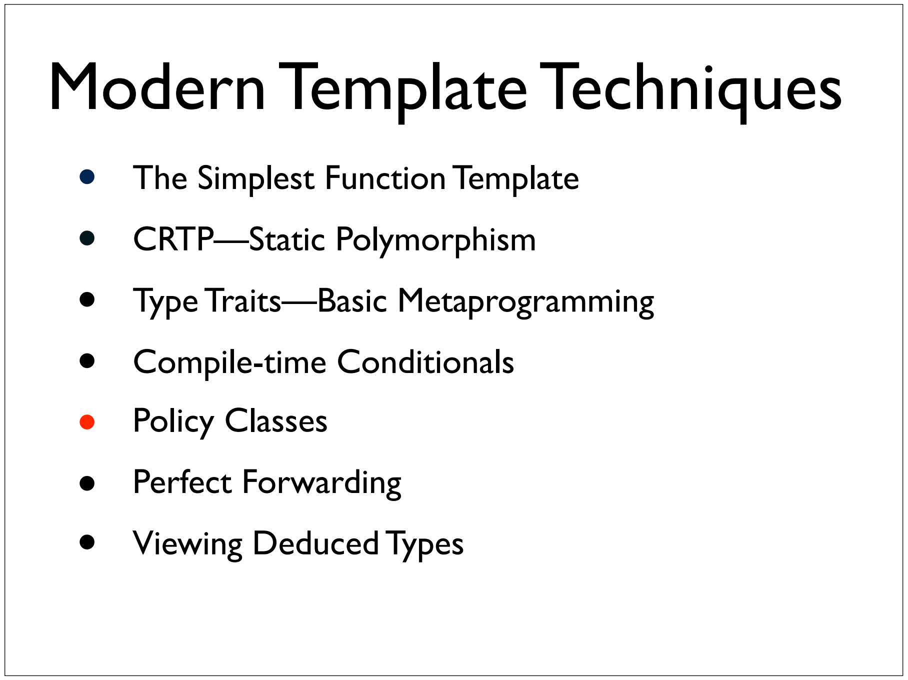
Policy Classes

Why can't we just use a virtual function?

The template solution requires no run-time overhead.

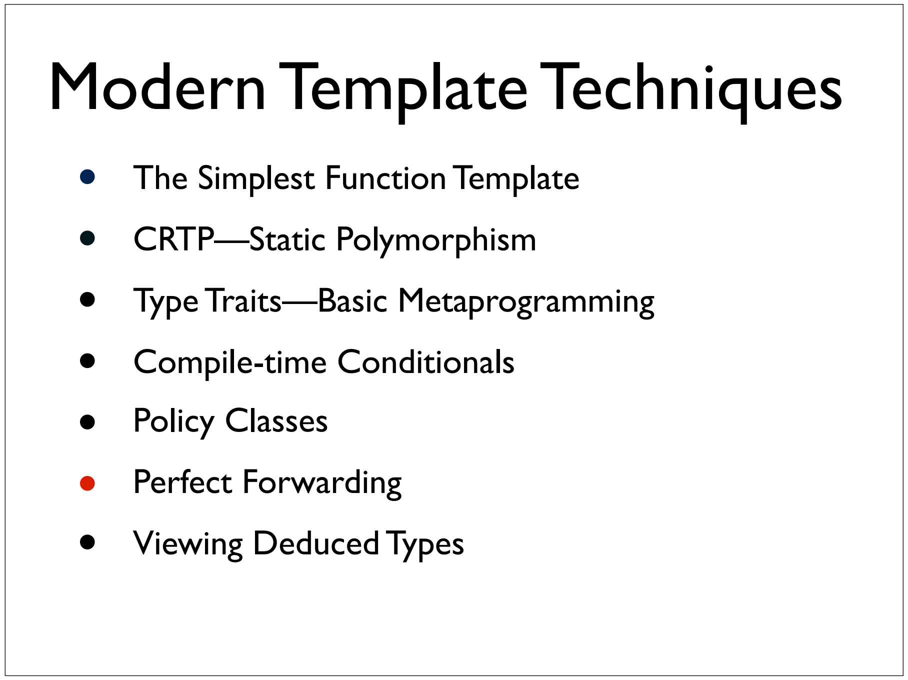
Consider the "do nothing" example. The static version compiles away to nothing.

But a virtual function implementation would require a non-inline-able dereferenced function call.



Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types



Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Perfect Forwarding

Sometimes we need to add a layer of functionality between two existing layers without modifying the data passed between the layers.

To do that we *perfect forward* the parameters passed to our function on to a target function.

Perfect Forwarding

Imagine that we've been give a library and we'd like to measure how much time we are spending in this library.

We create a class, APITimer, with a static data member that measures total time in library.

The class constructor starts a timer and the destructor stops it and adds the elapsed time to the static data member.

```
template <class APIFunction>
decltype(auto) TimeCall(APIFunction f)
{
  APITimer timer;
  return f();
}
```

Perfect Forwarding

```
template <class APIFunction>
decltype(auto) TimeCall(APIFunction f)
{
   APITimer timer;
   return f();
}
```

With "auto" this will compile in C++14 without a return type, but what if we where in C++11?

```
template <class APIFunction>
auto TimeCall(APIFunction f) -> decltype(f())
{
   APITimer timer;
   return f();
}
```

Perfect Forwarding

```
template <class APIFunction>
decltype(auto) TimeCall(APIFunction f)
{
   APITimer timer;
   return f();
}
```

What if the function has a parameter?

```
template <class APIFunction, class T>
decltype(auto) TimeCall(APIFunction f, T t)
{
   APITimer timer;
   return f(t);
}
```

Perfect Forwarding

What if the parameter is taken by reference to modify passed data?

```
template <class APIFunction, class T>
decltype(auto) TimeCall(APIFunction f, T& t)
{
   APITimer timer;
   return f(t);
}
```

What if the parameter is a temporary?

```
template <class APIFunction, class T>
decltype(auto) TimeCall(APIFunction f, T const& t)
{
   APITimer timer;
   return f(t);
}
```

Perfect Forwarding

We can go down the path of writing a separate overload for each scenario, but that doesn't scale in the face of multiple parameters each of which can be different in const/non-const, volatile/non-volatile, and rvalue/lvalue.

The solution is a *forwarding reference.*

AKA a *universal reference.*

Syntactically it is an rvalue reference, but in the context of type deduction (such as a template function instantiation) it can magically bind to either an rvalue or an lvalue parameter.

The parameter can be forwarded with `std::forward`.

Perfect Forwarding

```
template <class APIFunction, class T>
decltype(auto) TimeCall(APIFunction f, T&& t)
{
  APITimer timer;
  return f(std::forward<T>(t));
}
```

What if there are more than one parameter?

We could create an infinite number of versions, or we can use variadic template parameters.

```
template <class APIFunction, class... Args>
decltype(auto) TimeCall(APIFunction f, Args&&... args)
{
  APITimer timer;
  return f(std::forward<Args>(args)...);
}
```

Perfect Forwarding

```
template <class APIFunction, class... Args>
decltype(auto) TimeCall(APIFunction f, Args&&... args)
{
  APITimer timer;
  return f(std::forward<Args>(args)...);
}
```

Perfect forwarding is not really perfect (there are a few failure cases), but it is a valuable tool in the library builder’s toolkit.

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types

Viewing Deduced Types

Some when looking at code, we want to shake the compiler and say, “What is the type of *that*.”

- Using the Standard
- Using Boost
- Using the compiler

The Standard

#include <typeinfo>

typeid(x).name()

Will return a char const* that just *might* be helpful.

Assuming:

- You can build
- Don't care about const, volatile, and reference-ness
- You can parse the string: PK6Widget

Boost.TypeIndex

```
#include “boost/type_index.hpp"

using boost::typeindex::type_id_with_cvr;

type_id_with_cvr<T>.pretty_name()
type_id_with_cvr<decltype(param)>.pretty_name
```

Will return an std::string that include const, volatile, and references (_cvr) and will be less cryptic than the standard approach.

Compiler

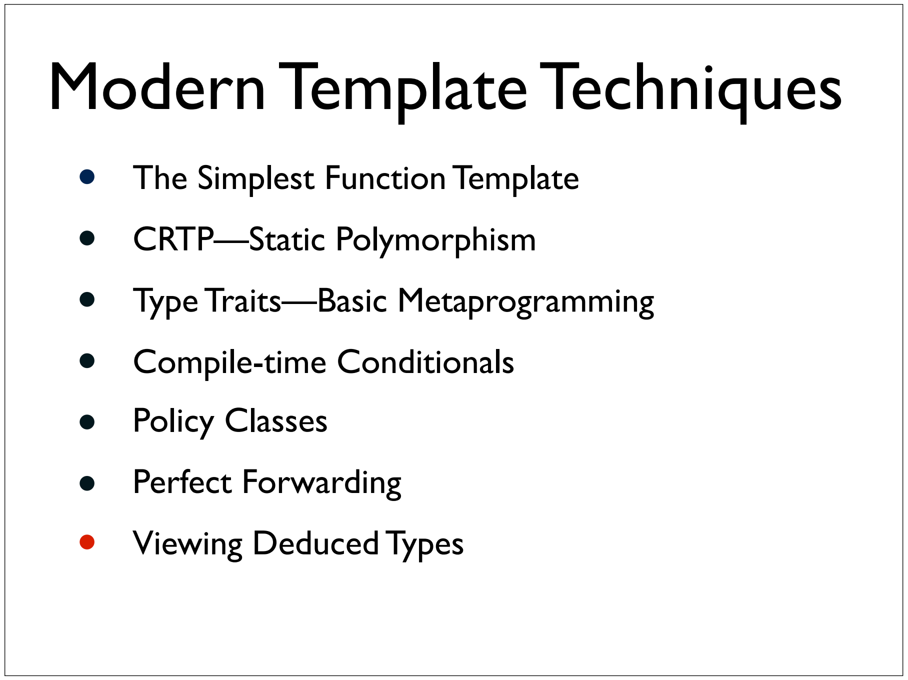
```
template<class T> class that_type;
template<class T> void name_that_type(T& param)
{
    that_type<T> tType;
    that_type<decltype(param)> paramType;
}
```

name_that_type(x)

Will generate error messages that will tell you both the type of T and the type of param:

error: implicit instantiation of undefined template ‘that_type<char>'

error: implicit instantiation of undefined template ‘that_type<char &>’



Modern Template Techniques

- The Simplest Function Template
- CRTP—Static Polymorphism
- Type Traits—Basic Metaprogramming
- Compile-time Conditionals
- Policy Classes
- Perfect Forwarding
- Viewing Deduced Types