Type Traits, Argument-Dependent Lookup; Templates II: CRTP, Variadic Templates, `decltype`

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Autumn 2020

Type Traits

- header `<type_traits>`
- utilities for compile-time querying and transformations of types
- `template< typename T, T v > struct integral_constant;`
  - compile-time constant of type `T`, with value `v`
  - defines static `constexpr` member `value`
  - `using true_type = integral_constant< bool, true >;`
  - `using false_type = integral_constant< bool, false >;`
- `TRAIT_v<...>` are helper variables equal to `TRAIT<...>::value` (since C++17)

Type Traits – Properties of Types

can be used to query information about types

- inherit from `true_type` or `false_type`
- `is_integral`, `is_arithmetic`, `is_same`, `is_array`, `is_rvalue_reference`, `is_trivially_default_constructible`, `is_nothrow_move_constructible`, ...
- binary: `is_same`, `is_convertible`, `is_base_of`

Type Traits – Properties of Types

can be used to query information about types

- inherit from `true_type` or `false_type`
- `is_integral`, `is_arithmetic`, `is_same`, `is_array`, `is_rvalue_reference`, `is_trivially_default_constructible`, `is_nothrow_move_constructible`, ...
- binary: `is_same`, `is_convertible`, `is_base_of`

usage

- static assert: prevent unwanted instantiations

```
template< typename BaseType >
struct Bignum {
  static_assert( std::is_integral< BaseType >::value,
                 "Bignum must be based on integral type" );
```

  - static assert checked at compile time
  - comment can be omitted in C++17

Type Traits – Tag Dispatch

- specialization based on type property
- takes advantage from the fact that query traits inherit from `std::true_type` or `std::false_type` and therefore are convertible to one of them

```
template< typename T >
void _constructRange( T *, T *, std::true_type ) { }
template< typename T >
void _constructRange( T *from, T *to, std::false_type ) {
    for ( ; from != to; ++from ) new ( from ) T();
}
template< typename T >
void constructRange( T *from, T *to ) {
  _constructRange( from, to,
      std::is_trivially_constructible< T >() );
}
```

Type Traits – Alternative Solution (C++17)

C++17 `if constexpr`

- `if` evaluated at compile time
- the other branch still needs to be syntactically valid
  - does not participate in return type deduction
  - if inside a templated entity, the other branch is not instantiated

```
using namespace std;
template< typename T >
void constructRange( T *from, T *to ) {
  if constexpr ( !is_trivially_constructible< T >::value )
    for ( ; from != to; ++from )
      new ( from ) T();
}
```

Type Traits – Type Transformations

- type functions that allow modification of types
- define nested typename `type`
  - `TRAIT_t< ... >` are aliases to `TRAIT< ... >::type` (since C++14)
- `remove_reference`, `add_lvalue_reference`, `remove_const`, `add_const`, `make_signed`, ...
- `std::conditional< b, T, F >` – defines member `type` to be `T` if `b` is `true` and to `F` otherwise

Type Decay

When a value `x` is passed to unrestricted template argument (i.e. `template< typename T > void foo( T );`), the derived type might not be exactly the type of `x`

- `T` will be value type (regardless of `x` being lvalue or rvalue)
- array will be transformed to pointer
- function type will be transformed to function pointer type
- `const` and `volatile` qualifiers (cv-qualifiers) are removed

Type Decay

When a value `x` is passed to unrestricted template argument (i.e. `template< typename T > void foo( T );`), the derived type might not be exactly the type of `x`

- `T` will be value type (regardless of `x` being lvalue or rvalue)
- array will be transformed to pointer
- function type will be transformed to function pointer type
- `const` and `volatile` qualifiers (cv-qualifiers) are removed

- decay trait – apply these conversions to a type
  - `decay_t< int >` ⇝ `int`
  - `decay_t< int && >` ⇝ `int`
  - `decay_t< const int & >` ⇝ `int`
  - `decay_t< int[100] >` ⇝ `int *`
  - `decay_t< int ( int ) >` ⇝ `int (*)( int )`

Argument-Dependent Lookup (ADL) I

```
foo( x, y, z ) // what does this call?
```

`foo` is unqualified (does not have a namespace), where will it be looked for?

1. `foo` called inside a function: the current block, and the blocks which contain it, until the function block is reached (inside-out)
2. `foo` is called in the definition of a member function: the class in which the member is declared, and its base classes
3. current namespace, and its enclosing namespaces up to the global one
   - all members of anonymous namespace behave as if they were defined in the enclosing non-anonymous namespace
   - the same works for members of `inline` namespaces
   - functions imported by `using bar::foo` or `using namespace bar` behave as if declared in the namespace containing the `using`

Argument-Dependent Lookup (ADL) I

```
foo( x, y, z ) // what does this call?
```

`foo` is unqualified (does not have a namespace), where will it be looked for?

1. `foo` called inside a function: the current block, and the blocks which contain it, until the function block is reached (inside-out)
2. `foo` is called in the definition of a member function: the class in which the member is declared, and its base classes
3. current namespace, and its enclosing namespaces up to the global one
   - all members of anonymous namespace behave as if they were defined in the enclosing non-anonymous namespace
   - the same works for members of `inline` namespaces
   - functions imported by `using bar::foo` or `using namespace bar` behave as if declared in the namespace containing the `using`

- **namespace of any of the arguments, of base classes of the arguments, or of template parameters of types of arguments**
  - same priority as those found above; except for `using`

Argument-Dependent Lookup Example

```
using std::cout;
namespace util {
  struct MyVector { /* ... */ };
  auto begin( MyVector &x ) { /* ... */ }
}
int main() {
  cout << "bla\n"; // calls std::operator<<( cout, "bla\n" )
  std::vector< int > vec;
  begin( vec ); // calls std::begin
  util::MyVector mv;
  begin( mv ); // calls util::begin
}
```

Argument-Dependent Lookup II

- ADL-imported functions can conflict with functions found using standard unqualified name lookup
  - they are all used for overload lookup at the same time
- ADL is important for working of stream operators <<, >>
  - you can overload them for your own class in your own namespace
- it is also why `swap` for unknown type (e.g. given by a template argument) should be written like this:

```
using std::swap;
swap( a, b );
```

  - this way if the namespace of `a`/`b` defines `swap` overload for their type it is used
  - otherwise `std::swap` is used
- it is actually a bit more complicated:
  http://en.cppreference.com/w/cpp/language/adl

Simplifying Code Through Templates

Situation: several classes that define some *basic* functionality differently, but further *derived* functionality is implemented the same way.

- `operator !=` can be implemented using `==`
- `rbegin`/`rend` can be implemented using `std::reverse_iterator` from `begin`/`end`
- duplicating the same code is tedious and error-prone

Simplifying Code Through Templates

Situation: several classes that define some *basic* functionality differently, but further *derived* functionality is implemented the same way.

- `operator !=` can be implemented using `==`
- `rbegin`/`rend` can be implemented using `std::reverse_iterator` from `begin`/`end`
- duplicating the same code is tedious and error-prone

possible solutions:

- use common base class and virtual member functions
  - several downsides: slower, works well only if the return type is always the same (i.e. works for `!=` but not for `rbegin` which depends on the type of `iterator`)
- use templates and *curiously recurring template pattern*
  - works always, runtime speed same as if implemented directly
  - compilation can be a bit slower

Curiously Recurring Template Pattern

```
template< typename Self > struct Eq {
    const Self& self() const {
        return static_cast< const Self& >( *this );
    }
    bool operator!=(const Self& other) const {
        return !(self() == other);
    }
};

struct Num : Eq< Num > {
    int value;
    friend bool operator==(const Num& a, const Num& b) {
        return a.value == b.value;
    }
};
```

Curiously Recurring Template Pattern

- works even with `friend` functions
  - thanks to ADL

```
template< typename Self > struct Eq {
    friend bool operator!=(const Self& a, const Self& b) {
        return !(a == b);
    }
};

struct Num : Eq< Num > {
    int value;
    friend bool operator==(const Num& a, const Num& b) {
        return a.value == b.value;
    }
};
```

Dependent Names

Names Inside Templates

- dependent vs. non-dependent
- names that depend on the template instantiation
- notably: `this` is a dependent name

Using `typename`

```
template< typename T >
void printVector( const std::vector< T >& v ) {
    typename std::vector< T >::size_type size = v.size();
    // does not work without typename, why?
    // ...
}
```

More info:

`http://en.cppreference.com/w/cpp/language/dependent_name`

Dependent Names

Using `this`

```
int x = 42;
template< typename T >
struct A : T {
    void f() { std::cout << x << ", "; }
    void g() { std::cout << this->x << ", "; }
};
struct X { int x = 17; };
struct Y {};

int main() {
    A<Y> ay;
    A<X> ax;
    ay.f(); ax.f(); ax.g();
}
```

Dependent Names

Using `this`

```
int x = 42;
template< typename T >
struct A : T {
    void f() { std::cout << x << ", "; }
    void g() { std::cout << this->x << ", "; }
};
struct X { int x = 17; };
struct Y {};

int main() {
    A<Y> ay;
    A<X> ax;
    ay.f(); ax.f(); ax.g();
}
```

Output: 42, 42, 17,

Refresh: Function Overloading

- more functions applicable to the given parameters
- need to consider priority
  1. function with less conversions needed wins
     - adding `const` to reference is also a conversion
  2. non-templated wins over templated
  3. more specialised template (with less template arguments) wins

Note: for more details, see

- `http://en.cppreference.com/w/cpp/language/overload_resolution`
- `http://en.cppreference.com/w/cpp/language/function_template#Function_template_overloading`

Templated Functions & Overloading

```
void f( int )  { std::cout << "int, "; }  // (1)
void f( long ) { std::cout << "long, "; } // (2)
template< typename T >
void f( T )    { std::cout << "T, "; }    // (3)

int main() {
    // what is the output?
    f( 42 );                 // (a)
    f( unsigned( 42 ) ); // (b)
    f( long( 42 ) );         // (c)
    f( short( 42 ) );        // (d)
}
```

Templated Functions & Overloading

```
void f( int )  { std::cout << "int, "; }  // (1)
void f( long ) { std::cout << "long, "; } // (2)
template< typename T >
void f( T )    { std::cout << "T, "; }    // (3)

int main() {
    // what is the output?
    f( 42 );                 // (a)
    f( unsigned( 42 ) ); // (b)
    f( long( 42 ) );         // (c)
    f( short( 42 ) );        // (d)
}
```

Output: `int, T, long, T,`

Templated Functions & Overloading

```
void g( int, int ) { std::cout << "ii, "; } // (1)
template< typename T >
void g( int, T )   { std::cout << "iT, "; } // (2)
template< typename T >
void g( T, int )   { std::cout << "Ti, "; } // (3)
template< typename T >
void g( T, T )     { std::cout << "TT, "; } // (4)
template< typename T, typename U >
void g( T, U )     { std::cout << "TU, "; } // (5)

int main() {
    g( 1, 1 ); g( 1, 1L ); g( 1L, 1 ); g( 1L, 1L );
    g( 1L, 1U ); g( 1U, 1U );
}
```

Templated Functions & Overloading

```
void g( int, int ) { std::cout << "ii, "; } // (1)
template< typename T >
void g( int, T )   { std::cout << "iT, "; } // (2)
template< typename T >
void g( T, int )   { std::cout << "Ti, "; } // (3)
template< typename T >
void g( T, T )     { std::cout << "TT, "; } // (4)
template< typename T, typename U >
void g( T, U )     { std::cout << "TU, "; } // (5)

int main() {
    g( 1, 1 ); g( 1, 1L ); g( 1L, 1 ); g( 1L, 1L );
    g( 1L, 1U ); g( 1U, 1U );
}
```

Output: `ii, iT, Ti, TT, TU, TT,`

Templated Function & Specialisation

```
void f( int )   { std::cout << "int, "; } // (1)
template< typename T >
void f( T )     { std::cout << "T, "; }   // (2)
template<>
void f<>( int ) { std::cout << "T:i, "; } // (3)

int main() {
    f( 1 ); f( long( 1 ) ); f<>( 1 );
}
```

Templated Function & Specialisation

```
void f( int )    { std::cout << "int, "; } // (1)
template< typename T >
void f( T )      { std::cout << "T, "; }   // (2)
template<>
void f<>( int ) { std::cout << "T:i, "; } // (3)

int main() {
    f( 1 ); f( long( 1 ) ); f<>( 1 );
}
```

Output: `int, T, T:i`

- specialization used only after the template it specializes was selected, it is not part of overload resolution
- **Recommendation:** do not use template specialisation for *functions*; see http://www.gotw.ca/publications/mill17.htm

Selecting Overload by Property

- sometimes it is useful to be able to overload based on some property of the argument types
- tag dispatch or `constexpr if` is a simple solution, but cannot be always used
  - not really overloaded, or overloading is hidden (in case of tag dispatch)
  - there has to be a trait/boolean predicate for the property

Selecting Overload by Property

- sometimes it is useful to be able to overload based on some property of the argument types
- tag dispatch or `constexpr if` is a simple solution, but cannot be always used
  - not really overloaded, or overloading is hidden (in case of tag dispatch)
  - there has to be a trait/boolean predicate for the property
- more general mechanism needed
  - example: an overload should be used if the argument's type defines some nested typename
  - example: an overload should be used if the argument is a container (defines `begin` and `end`)
- idea: allow hiding of some overloads based on some condition

Substitution Failure is not an Error (SFINAE)

- an error in substitution of type variable (for example missing nested typename) in a *function header* need not lead to compilation error
- the overload with substitution failure is ignored
- other overloads might be used

```
struct A { using Foo = int; };
struct B { using Bar = int; };
template< typename T >
typename T::Foo foo( T ) {} // (1)
template< typename T >
typename T::Bar foo( T ) {} // (2)

int main() {
    foo( A() ); // calls (1)
    foo( B() ); // calls (2)
}
```

Derived Operators Using SFINAE

```
struct Eq { using IsEq = bool; };

struct Foo : Eq {
    bool operator==( const Foo &o ) const
    { /* ... */ }
}

template< typename T >
typename T::IsEq operator!=( const T &a, const T &b )
{ return !(a == b); }
```

- uses namespace-level operator `!=` which is usable only for types which define `IsEq`
- cleaner than CRTP, `Eq` is just a tag
- not usable for member functions
- works even if `Eq` is in a namespace (thanks to ADL)

Enabling Overloads Based on Boolean Condition

- example conditions: argument type is integral, index is in the range of `std::array`/`std::tuple`,...
- using a type which has a nested typename if a condition holds
- `template< bool cond, typename T > std::enable_if;`
  - defines `type` to `T` if `cond` is `true`
  - defines *no* `type` *member* if `cond` is `false`
  - `T` is defaulted to `void`

Enabling Overloads Based on Boolean Condition

- example conditions: argument type is integral, index is in the range of `std::array`/`std::tuple`,...
- using a type which has a nested typename if a condition holds
- `template< bool cond, typename T > std::enable_if;`
  - defines `type` to `T` if `cond` is `true`
  - defines *no* `type` *member* if `cond` is `false`
  - `T` is defaulted to `void`

```
template< size_t I, typename T, size_t N >
auto getWDef( const std::array< T, N > &, const T &def )
    -> std::enable_if_t< (I >= N), T >
{ return def; }

template< size_t I, typename T, size_t N >
auto getWDef( const std::array< T, N > &arr, const T & )
    -> std::enable_if_t< (I < N), T >
{ return std::get< I >( arr ); }
```

Better Operators Using enable_if

```
struct Eq { };
struct Ord : Eq { };

template< typename T, typename Trait, typename R >
using IsTrait = std::enable_if_t<
                    std::is_base_of< Trait, T >::value, R >;

template< typename T, typename R = bool >
using IsEq = IsTrait< T, Eq, R >;

template< typename T >
IsEq< T > operator!=( const T &a, const T &b )
{ return !(a == b); }
```

- more in `operator.cpp`
- same principle can be used for arithmetic operators, iterator operators

Variadic Templates

templates that have a variable number of arguments

Syntax

```
// variadic class template
template< typename ... Args >
class X { /* ... */ };

// variadic function template
template< typename ... Args >
void foo( Args ... args ) { /* ... */ };

// does not have to be just types
template< int ... Nums >
void foo() { /* ... */ };
```

- since C++17 we can also use `template< auto ... Args >`
- `sizeof...( Args )` returns number of arguments

Working With Argument Packs

- forward somewhere else
- recursion
- fold expressions (C++17)

```
template< typename ... Args >
void foo( Args ... args ) {
    do_something( args... );
    // expands to: do_something( arg1, arg2, ..., argN );
    SomeClass obj{ args... };
    // expands to: SomeClass obj{ arg1, arg2, ..., argN };
    do_something( modify(args)... );
    // expands to: do_something( modify(arg1),
    //       modify(arg2), ..., modify(argN) );
}
```

- beware: evaluation order of function arguments is not specified, while evaluation order inside braces {} is left-to-right

Forwarding Argument Packs

```
template< typename T, typename ... Args >
std::unique_ptr< T > make_unique( Args ... args ) {
    return std::unique_ptr< T >( new T( args... ) );
}
```

- what's wrong (inefficient)?

Forwarding Argument Packs

```
template< typename T, typename ... Args >
std::unique_ptr< T > make_unique( Args ... args ) {
    return std::unique_ptr< T >( new T( args... ) );
}
```

- what's wrong (inefficient)?

Perfect Forwarding

```
template< typename T, typename ... Args >
std::unique_ptr< T > make_unique( Args && ... args ) {
    return std::unique_ptr< T >(
        new T( std::forward< Args >( args )... )
    );
}
```

Expanding Argument Packs With Recursion I

```
void print() { std::cout << std::endl; }

template< typename T, typename ... Args >
void print( const T & t, const Args & ... args ) {
    std::cout << t;
    print( args... );
}

int main() {
    print( "Hello, ", ' ', "world! ", 42 );
}
```

Expanding Argument Packs With Recursion II

```
template< typename T >
const T & max( const T & t ) { return t; }

template< typename T, typename ... Args >
const T & max( const T & a, const T & b,
               const Args & ... args ) {
    return std::max( a, max( b, args... ) );
}

int main() {
    return max( 1, 7, 4, 12, 17, 42, 0, -8 );
}
```

Fold Expressions for Variadic Templates

- C++17
- allow transforming of variadic packs
- similar to functional programming folds or `std::accumulate`
- `http://en.cppreference.com/w/cpp/language/fold`

Fold Expressions for Variadic Templates

- C++17
- allow transforming of variadic packs
- similar to functional programming folds or `std::accumulate`
- http://en.cppreference.com/w/cpp/language/fold

```
template< typename... Args > // unary left fold
bool all( Args ... args ) { return (... && args); }

template< typename ...Args > // binary left fold
void println( const Args &... args ) {
    (std::cout << ... << args) << '\n';
}
template< typename T, typename ... Args> // unary right
void push_back_vec(std::vector<T>& v, Args&&... args)
{ ( v.push_back( std::forward< Args >( args ) ), ...); }
```

- order of evaluation in the last case is guaranteed – left to right
  - unless `operator,` is overloaded

`decltype` in the Return Type

sometimes we need to get type from an expression at compile time

```
template< typename X, typename Y >
auto plus( X &&x, Y &&y ) -> decltype( x + y ) {
    return x + y;
}
```

- the return type of `plus` is the same as the type of `x + y`
- `x` and `y` has to be defined, so we must use trailing return type
- if `x + y` is not a valid expression (there is no matching `operator+`) SFINAE disables this function (overload)
- since C++14 the return type can be omitted to detect it from `return` statement(s)
  - not equivalent to the above example – SFINAE does not work

`decltype` in General

```
struct X { long x; };
int main( void ) {
    decltype( 1 ) x = 1; // x is int
    decltype( x ) y = 2; // y is int
    // beware of double parentheses
    decltype( ( x ) ) z = x; // z is int & (!)
    X obj;
    decltype( obj.x ) t = 3; // t is long
}
```

- derive the type from a named entity (variable, class member, . . .)
  - the exact declared type of the entity
- derive the type from an expression
  - lvalue reference if the expression is an lvalue
- http://en.cppreference.com/w/cpp/language/decltype

Using `decltype` in SFINAE

```
template< typename C > // (1)
auto begin( C &c ) -> decltype( c.begin() );
template< typename C > // (2)
auto begin( const C &c ) -> decltype( c.begin() );
template< typename T, size_t N > // (3)
T *begin( T (&array)[N] );
```

- for an object `c` with a `begin` method, either `(1)` or `(2)` is used
  - substitution fails for `(3)`
- for a static array, `(3)` is used
  - substitution fails for `(1)` and `(2)`
- in all other cases substitution fails for all overloads and causes compilation failure
- no overloading conflicts

Overloading Conflicts

- happens when two overloads are available and they are "equally good"
  - often present with SFINAE

```
template< typename T >
auto toStringAny( T &&x )
    -> decltype( std::to_string( x ) )
{ return std::to_string( x ); }

template< typename T >
std::string toStringAny( T && )
{ return "<<not printable>>"; }
```

- if the `decltype` succeeds there is no way to resolve the overloads

Overloading Conflicts: Resolution

```
struct Preferred { };
struct NotPreferred { NotPreferred(Preferred) {} };

template< typename T > // (1)
auto _toStringAny( T &&x, Preferred )
    -> decltype( std::to_string( x ) )
{ return std::to_string( std::forward< T >( x ) ); }

template< typename T > // (2)
std::string _toStringAny( T &&x, NotPreferred )
{ return "<<not printable>>"; }

template< typename T >
std::string toStringAny( T &&x ) {
    return _toStringAny( std::forward< T >( x ),
                         Preferred() );
}
```

Overloading Conflicts: Resolution

- version `(2)` requires a cast (from `Preferred` to `NotPreferred`)
- therefore version `(1)` takes precedence if both are enabled
- a nicer version of the common `int`/`unsigned` trick
- more overloads require more additional parameters

`declval` Helper

- get a value placeholder for given type
- `std::declval< T >()` returns a r-value reference to `T`
  - use `std::declval< T & >()` to get l-value reference
- works only in `decltype` (not-evaluated) context

```
#include <utility>
struct Bar { /* ... */ };
template< typename Fn >
auto call( Fn fn )
    -> decltype( fn( std::declval< Bar & >() ) )
{ /* ... */ }
```

`declval` Helper

- get a value placeholder for given type
- `std::declval< T >()` returns a r-value reference to `T`
  - use `std::declval< T & >()` to get l-value reference
- works only in `decltype` (not-evaluated) context

```
#include <utility>
struct Bar { /* ... */ };
template< typename Fn >
auto call( Fn fn )
    -> decltype( fn( std::declval< Bar & >() ) )
{ /* ... */ }
```

- but `std::result_of_t< Fn( Bar ) >` would be better solution here (it uses `declval` internally)

SFINAE in Constructors and Cast Operators

- constructor and cast operators do not have return type, where to put SFINAE?

SFINAE in Constructors and Cast Operators

- constructor and cast operators do not have return type, where to put SFINAE?
- use template parameters with a default

```
#include <type_traits>
template< typename T >
struct SmartPtr {
  template< typename Y,
    typename = decltype( static_cast< T * >(
                  std::declval< Y * >() ) ) >
  explicit SmartPtr( Y *ptr ) { /* ... */ }
};
```

- overloads cannot differ only in the `template` specification