# Templates II

## PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

# Outline

- Type Traits
- Curiously Recurring Template Pattern (CRTP)
- Substitution Failure Is Not An Error (SFINAE)

# 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 >;`
- properties of 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`, ...

# Type Traits

type functions/transformations

- define `type` nested typename
  - `*TRAIT_t` are aliases to `TRAIT< ... >::type` (C++14)
- unary transformations: `remove_reference`, `add_lvalue_reference`, `remove_const`, `add_const`, `make_signed`, ...
- binary: `is_same`, `is_convertible`, `is_base_of`
- `decay` – apply conversions that would be applied if the value of given type were passed by value to a function (which takes an unconstrained templated argument)
  - `decay_t< int >` $\leadsto$ `int`
  - `decay_t< int && >` $\leadsto$ `int`
  - `decay_t< const int & >` $\leadsto$ `int`
  - `decay_t< int[100] >` $\leadsto$ `int *`
  - `decay_t< int ( int ) >` $\leadsto$ `int (*)( int )`

# Type Traits: Usage

**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

# Type Traits: Usage

**tag dispatch** (specialization based on type property)

```
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 >() );
}
```

# Motivation: Operators

In C++ when we define `operator==` we should also define `operator!=`, similarly for all ordering operators, numeric operators, ...

- other could often be derived automatically
  - derive != from ==; >, <=, >= from < and ==; + from +=; ...
  - C++ can't do this automatically
  - manual definitions are error prone
- we want this to work statically (without virtual calls)

# Curiously Recurring Template Pattern (CRTP)

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

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

- Foo has both `==` and `!=`
- the base statically knows its derivative, so it can access it
- can be used to inject any member function into a class

# Overloading by Property

- sometimes it is usable to be able to overload based on some property of the argument types
- tag dispatch is simple solution, but cannot be always used
  - hides the fact that the function is overloaded (using helpers)
  - there must be a trait for the property
  - 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 >
auto foo( T ) -> typename T::Foo {} // (1)
template< typename T >
auto foo( T ) -> typename T::Bar {} // (2)

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

# Operators Using SFINAE

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

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

template< typename T >
auto operator!=( const T &a, const T &b ) ->
    typename T::IsEq
{ 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

# Argument-Dependent Lookup (ADL)

- what if `Eq` from previous example is defined in one namespace and `Foo` in different namespace?

# Argument-Dependent Lookup (ADL)

- what if Eq from previous example is defined in one namespace and Foo in different namespace?

- it still works thanks to Argument-Dependent Lookup!
- function lookup looks into namespaces of the arguments and their predecessors
- http://en.cppreference.com/w/cpp/language/adl

```
namespace util {
    struct MyVector { /* ... */ };
    auto begin( MyVector &x ) { /* ... */ }
}
int main() {
    std::vector< int > vec;
    begin( vec ); // calls std::begin
    util::MyVector mv;
    begin( mv ); // calls util::begin
}
```

# Enabling Overloads Based on Boolean Condition

- example conditions: argument type is integral, index is in the range of `std::array`/`std::tuple`,...
- can be done by using a type which has nested typename if a condition holds
- `template< bool cond, typename T > std::enable_if;`
  - defines type to `T` if `cond` is `true`
  - `T` is defaulted to `void`

```
template< size_t I, typename T, size_t N >
auto getWDef( const std::array< T, N > &, const T &def )
    -> typename std::enable_if< (I<0 || I>=N), T >::type
{ return def; }

template< size_t I, typename T, size_t N >
auto getWDef( const std::array< T, N > &arr, const T & )
    -> typename std::enable_if< (I>=0 && I<N), T >::type
{ return std::get< I >( arr ); }
```

# SFINAE With Tuples

```
#include <type_traits> // std::enable_if
#include <tuple>
template< int i, int c, class Os, class... Args >
auto _fmt_t( Os &os, const std::tuple<Args...> &t )
    -> typename std::enable_if< (i >= c) >::type { }

template< int i, int c, class Os, class... Args >
auto _fmt_t( Os &os, const std::tuple<Args...> &t )
    -> typename std::enable_if< (i < c) >::type
{
    os << std::get< i >( t );
    _fmt_t< i + 1, c >( os, t );
}
template< class Os, class... Args >
void fmt_tuple( Os &os, const std::tuple<Args...> &t )
{ return _fmt_t< 0, sizeof...(Args) >(os, t); }
```

# Task

1. Create `Ord` tag which works similarly as `Eq` but allows deriving of relational operators from == and <
   - deriving from `Ord` implies usage of `Eq`
2. Extend the `vector` from previous lectures so that it calls constructor and destructors only if the constructor (destructor) is not trivial