Templates III

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Outline

- **decltype**, more SFINAE
- overload resolution ordering
- **constexpr**
- template template parameters

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)
- in 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 x;
    decltype( x.x ) z = 3 // z is long
}
```

- derive the type from an expression (at compile time)
  - for an object name it is the type of the object
  - unless it is in parentheses, then it is lvalue of the type
- not widely used
- `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 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 & >() ) )
{ /* ... */ }
```

SFINAE in Constructors and Cast Operators

- does 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

constexpr

```
constexpr long fact( long x ) {
    return x == 0 ? 1 : x * fact( x - 1 );
}
template< long X > struct Test {
    constexpr long get() const { return X; }
};
Test< fact( 16 ) > t;
int main( int argc, char **argv ) {
    std::cout << t.get() << std::endl;
    std::cout << fact( argc - 1 ) << std::endl;
}
```

- functions that can be evaluated at compile time
  - but also at runtime, if the arguments are not constant
- in C++11 it can contain only a single `return` statement
  - in C++14 more relaxed (no exceptions, allocation, calling non-constexpr functions, . . .)

Template Template Parameters

type template without parameters is not a type

- but it can be passed as a template parameter into another template

```
template< typename T > struct Wrapper { };
template< template< typename > class W, typename T >
struct Foo {
    W< T > val;
};
Foo< Wrapper, int > x; // x.val is Wrapper< int >
```

- has to specify the number and kind of template parameters
  - `typename`, value, `template`
  - can be variadic (matches template with any number of parameters of given kind)
- the only case of template specification where `typename` is not equivalent to `class` (up to C++17)

Fold Expressions for Variadic Templates

- C++17 (use `-std=c++1z` in a very new clang/gcc)
- allow transforming of variadic packs
- similar idea as functional programming folds or `std::accumulate`
- `http://en.cppreference.com/w/cpp/language/fold`

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

template< typename ...Args > // binary left fold
void printer( 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( args ), ...); }
```

Task: `constexpr`

Define a `constexpr` function for the calculation of the nth Fibonacci number

- use both C++11 (recursive) and C++14 (iterative) **`constexpr`**
- test that the value is available at the compile time (like in the previous example)

$$fib(n) = \begin{cases} 0 & \text{when } n = 0 \\ 1 & \text{when } n = 1 \\ fib(n-2) + fib(n-1) & \text{otherwise} \end{cases}$$