

Šablony podrobně III: metaprogramování

PB173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

8. prosince 2014

Šablony podrobně III

Na co se podíváme?

- `decltype`, SFINAE podruhé
- řízení pořadí přetížených variant funkce
- `constexpr`

decltype v návratovém typu

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

- návratový typ funkce plus je stejný jako typ výrazu $x + y$ pro dané typy X , Y
- aby bylo x , y definované musíme použít syntax s návratovým typem na konci
- navíc, pokud $x + y$ nelze provést (neexistuje `operator+` kompatibilní s typy), pak se tělo funkce nekompile a funkce je vynechána při řešení přetěžování (SFINAE)

decltype obecně

```
struct X { long x; }
```

```
int main( void ) {  
    decltype( 1 ) x = 1; // x je typu int  
    decltype( x ) y = 2; // y je typu int  
    X x;  
    decltype( x.x ) z = 3 // z je typu long  
}
```

- v době kompilace odvodí typ výrazu, který je jeho parametrem
- nepříliš používané v této formě
- <http://en.cppreference.com/w/cpp/language/decltype>

decltype: SFINAE

```
template< typename C > // (1)
auto begin( C & ) -> decltype( c.begin() );

template< typename C > // (2)
auto begin( const C & ) -> decltype( c.begin() );

template< typename T, size_t N > // (3)
T *begin( T (&array)[N] );
```

- pokud je parametrem objekt, s metodou begin() použije se (1) nebo (2)
 - (3) není možné použít, hlavička funkce je nesprávně utvořená (selhání substituce)
- pokud je parametr statické pole, použije se (3)
 - selhání substituce u (1) a (2)
- jinak selže substituce u všech verzí \Rightarrow chyba kompilace
- nedojde ke konfliktu mezi (1), (2) a (3)

Problém nejednoznačnosti s decltype

```
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>>";
}
```

- nejednoznačné co se má zavolat pokud je `std::to_string(x)` platné (`toStringAny(1),...`)
- nelze jednoduše napsat výraz, který je platný právě pokud je `std::to_string(x)` neplatné

Řešení nejednoznačnosti s decltype

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

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

Řešení nejednoznačnosti s `decltype`

- verze (1) má přednost pokud je aktivní, protože (2) vyžaduje konverzi
- pokud potřebujeme rozlišit mezi více variantami, musíme mít více Preferred parametrů

declval

- umožňuje získat „hodnotu“ daného typu pro použití v `decltype`
- `std::declval< T >()`: r-value reference na hodnotu typu T
- někdy vhodné explicitně použít l-value referenci
- při pokusu o vyhodnocení způsobí chybu kompilace

```
template< typename T >
auto print( const T &x ) -> decltype( void(
    std::declval< std::ostream & >() << x ) )
{
    // ...
}
```

SFINAE a konstruktory

- typické je používat pro ověření substitute návratový typ
- konstruktory ale nemají návratový typ

Využijeme substitute v šablonovém parametru s výchozí hodnotou:

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

Pozor: nefunguje pokud chceme více přetížení lišících se jen v podmínce.

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;  
}
```

- constexpr funkce se vyhodnotí za kompilace
- lze volat i za běhu jako běžnou funkci
- může obsahovat jen **return** výraz (< C++14)

constexpr cvičení

Definujte `constexpr` funkci for výpočet fibonacciho čísla

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

Ověřte, že funguje za kompilace obdobně jako v předchozím příkladě.

Serializace do stringu.