Šablony podrobně I: přetěžování funkcí

PB173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

24. listopadu 2014

Motivace

„There are only two kinds of languages: the ones people complain about and the ones nobody uses.“

– Bjarne Stroustrup

„C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.“

– Bjarne Stroustrup

„Na světě není jediný programátor, který by mohl říct, že zná kompletně C++ a měl pravdu“

– neznámý autor

Šablony podrobně I

Na co se podíváme?

- jak se kompilují šablony
- pravidla/priority pro volání přetížených funkcí (za přítomnosti šablon)
- specializace funkcí
- odvozování typů parametrů

Kompilace šablon

Šablony se kompilují až s konkrétními šablonovými parametry (instanciace šablon)

- proto musí mít tělo v `.h`, pokud jsou tam deklarované
- (většina) typové kontroly probíhá při instanciaci
- šablona nemusí fungovat pro libovolné šablonové parametry

```
template< typename T >
T foo( T a, T b ) { return a + b; }
struct A {};

int main() {
    foo( 1, 1 ); // kompilace OK
    foo< std::string >( "a", "b" ); // OK
    // foo( A(), A() ); // nezkompiluje se
}
```

Priority přetížených funkcí

```
namespace std {
    string to_string( int );
    string to_string( long );
    string to_string( long long );
    string to_string( unsigned );
    string to_string( unsigned long );
    string to_string( unsigned long long );
    string to_string( float );
    string to_string( double );
    string to_string( long double );
}
```

Která se zavolá pro `to_string( 1 )`, `to_string( 3.14 )`, `to_string( 42ull )`?

Priority přetížených funkcí

```
template< typename T > struct shared_ptr {
    template< typename Y >
    explicit shared_ptr( Y * );          (1)
    shared_ptr( std::nullptr_t );        (2)
    shared_ptr( shared_ptr && );         (3)
    template< typename Y >
    shared_ptr( shared_ptr< Y > && );    (4)
    // ...
};
struct A { }; struct B : A { };
int main() {
    shared_ptr< B > pb1( new B() );
    shared_ptr< B > pb2( make_shared< B >() );
    shared_ptr< A > pa1( make_shared< B >() );
    shared_ptr< A > pa2( nullptr );
}
```

Priority přetížených funkcí

proč?

může být více funkcí, které se dají zavolat s danými argumenty

- různé funkce pro typy, které lze konvertovat
- šablonovaná funkce a konkrétní případ
- specializace šablon

musí být jasné, které mají nejvyšší prioritu

- pokud má více funkcí nejvyšší prioritu je to chyba při kompilaci

pravidla jsou rozsáhlá, zde jen to nejdůležitější, více viz `http://en.cppreference.com/w/cpp/language/overload_resolution`

Priority přetížených funkcí

pořadí

1. pokud některá funkce má méně nutných konverzí parametrů pak vítězí
2. jinak, pokud je jedna šablona a druhá není, má ne-šablona přednost
3. jinak, pokud je jedna ze šablon více specializovaná pak vítězí
   - zjednodušeně řečeno, čím víc šablonových parametrů tím později funkce přijde na řadu
   - pro odvážné: `http://en.cppreference.com/w/cpp/language/function_template#Function_template_overloading`

Funguje stejně pro (globální) funkce i metody tříd.

Priority přetížených funkcí

příklad: `overload_order_1.cpp`

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

Co vypíše?

```
f( 42 );                 // (a)
f( unsigned( 42 ) ); // (b)
f( long( 42 ) );         // (c)
f( short( 42 ) );        // (d)
```

Priority přetížených funkcí

příklad: `overload_order_1.cpp`

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

Co vypíše?

```
f( 42 );                 // (a)
f( unsigned( 42 ) ); // (b)
f( long( 42 ) );     // (c)
f( short( 42 ) );    // (d)
```

`int, T, long, T`

V (a) a (c) má (1), respektive (2) přesnou shodu a tedy přednost před šablonou (3). V (b) a (d) ale (1) i (2) vyžadují konverzi a tedy má (3) přednost.

Priority přetížených funkcí

složitější příklad: `overload_order_2.cpp`

```
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 X >
void g( T, X ) { std::cout << "TX,"; }     // (5)

g( 1, 1 ); g( 1, 1L ); g( 1L, 1 ); g( 1L, 1L );
g( 1L, 1u ); g( 1u, 1u );
```

Priority přetížených funkcí

složitější příklad: `overload_order_2.cpp`

```
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 X >
void g( T, X ) { std::cout << "TX,"; }     // (5)

g( 1, 1 ); g( 1, 1L ); g( 1L, 1 ); g( 1L, 1L );
g( 1L, 1u ); g( 1u, 1u );
```

`ii, iT, Ti, TT, TX, TT,` (4) je specializovanější než (5)

Specializace šablon funkcí 

`template_spec.cpp`

```
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)
```

Nejednoznačné?

Specializace šablon funkcí 

`template_spec.cpp`

```
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)
```

Nejednoznačné? Ve skutečnosti ne, (3) se zavolá jedině pro `f< int >( ... )`, ale nikdy pro `f( ... )`.

Při řešení přetěžování se specializace ignorují, na řadu přijdou až po vybrání příslušné šablony.

Specializace šablon funkcí 

Nepoužívat!

Z `http://www.gotw.ca/publications/mill17.htm`, kde je více podrobností.

```
template< typename T > // (a)
void f( T );
template<>             // (b) specializace (a)
void f<>( int * );
template< typename T > // (c) přetížení (a)
void f( T * );

int *p;
f( p );                // zavolá (c)!
```

Při řešení přetěžování se specializace ignorují, na řadu přijdou až po vybrání příslušné šablony.

Odvozování typů parametrů šablonové funkce

```
template< typename T, typename X >
void foo( T t, X x, X y );
```

- lze volat i bez explicitně zadaných typů parametrů, nebo s částečně zadanými:

```
foo(1,3,4);                 // T = int,  X = int
foo<>(1,2,3)                // T = int,  X = int
foo<long>(1,3,4);           // T = long, X = int
foo<long,long>(1,3,4); // T = long, X = long
```

- verze bez parametrů a <> je preferovaná
- typy nevyplněných parametrů se odvodí od typů argumentů, které jim odpovídají
- `foo( 1, long( 3 ), short( 4 ) )` je špatně: konflikt odvozených typů pro `X` (**long** vs. **short**)

Odvozování typů parametrů šablonové funkce

Odvozování funguje pokud

- se šablonový parametr nachází v typu některého parametru funkce

```
template< size_t I, class T, size_t N >
T &get( std::array< T, N > & );
```

(`T`, `N` lze odvodit, `I` nikoli)

- je poskytnuta výchozí hodnota

```
template< typename T = int >
T getRandom();
```

lze volat:

```
getRandom()          // T = int
getRandom<>()        // T = int
getRandom< long >()  // T = long
```

Odvozování typů parametrů šablonové funkce

Odvozený typ podrobně: reference

```
template< typename T > void foo( T );    // (1)
template< typename T > void bar( T & );  // (2)
template< typename T > void baz( T && ); // (3)
int main() {
    int x = 5;
    foo( x );   // void foo< int >( int )
    foo( 5 );   // void foo< int >( int )

    bar( x );   // void bar< int >( int & )
    bar( 5 );   // chyba: 5 je rvalue

    baz( x );   // void baz< int & >( int & )
    baz( 5 );   // void baz< int >( int && )
}
```

Odvozování typů parametrů šablonového objektu

Jedině za pomoci výchozí hodnoty:

```
template< typename T,
    typename Allocator = std::allocator<T> >
class vector;

// Allocator = std::allocator< int >
std::vector< int > v1;

// Allocator = MyAllocator< int >
std::vector< int, MyAllocator< int > > v2;
```

- <> nelze vynechávat:

```
template< typename T = int > struct Foo;
Foo<> f; //
Foo f2;  // compile error
```

Úloha

Jednoduchá knihovna pro formátování základních datových typů a kolekcí.
viz `http://cecko.eu/public/pb173/cviceni_10`