

# Šablony podrobně II: metaprogramování

## PB173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

1. prosince 2014

# Šablony podrobně II

Na co se díváme?

- omezení opakujícího se kódu, mixin
- řízení výběru přetížené alternativy, SFINAE, `<type_traits>`
- argument-dependent lookup

# Opakování kódu: operátory

Znáte z PB161:

```
struct X {  
    int a, b;  
};
```

```
bool operator==( const X &x, const X &y ) {  
    return x.a == y.a && x.b == y.b;  
}
```

```
bool operator!=( const X &x, const X &y ) {  
    return !(x == y);  
}
```

Operátor != bude vypadat vždy stejně. Podobně pro operátory < a <=, >, >= nebo += a +...

# Opakování kódu: operátory

- je typické implementovat některé operátory pomocí jiných: zajištění konzistence, méně opakování kódu
- avšak tato pomocná implementace je vždy stejná, liší se jen typem třídy
- C++ neumí automaticky vygenerovat zbylé operátory

# Opakování kódu: operátory

- je typické implementovat některé operátory pomocí jiných: zajištění konzistence, méně opakování kódu
- avšak tato pomocná implementace je vždy stejná, liší se jen typem třídy
- C++ neumí automaticky vygenerovat zbylé operátory

## Pozorování

- dědičnost v C++ neslouží jen k vytváření objektových hierarchií
  - lze ji využít i pro metaprogramování: modifikace/rozšíření programu v okamžiku kompilace
- ⇒ Mixin / Curriously Recurring Template Pattern

# Operátory: Mixin

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

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

# Mixin

- funkcionalita implementačně nezávislá na typu je vytažena do speciální třídy
- tato třída neslouží k samostatnému použití
- je pouze „přimíchána“ (*mix-in*) ke konkrétním třídám

## Implementace

- mixin zdědíme
- typicky parametrizujeme mixin typem koncové třídy
  - velikost mixinu nesmí záviset na koncové třídě
  - z mixinu lze získat koncovou třídu (`static_cast`)

## Výhody

- zabráníme duplikaci kódu
- statické vázání při kompilaci
  - na rozdíl od dynamické dědičnosti s `virtual`

# Typové predikáty a modifikátory

Predikáty a modifikátory nad typy (<type\_traits>)

[http://en.cppreference.com/w/cpp/header/type\\_traits](http://en.cppreference.com/w/cpp/header/type_traits)

- *při kompilaci* můžeme detekovat vlastnosti a vztahy typů
- predikáty
  - `std::is_integral< T >::value`
  - `std::is_pointer< T >::value`
  - `std::is_same< A, B >::value`
- typové modifikátory („typové funkce“)
  - `std::remove_reference< T >::type`
  - `std::add_const< T >::type`

```
template< typename T >
T foo( T t ) {
    static_assert( std::is_integral< T >::value,
                  "T is not integral" );

    // ...
}
```



# Typové predikáty a modifikátory

*// zjištění návratového typu*

```
template< typename Fn, typename A, typename B >  
auto apply( Fn &&fn, std::pair< A, B > &pair ) ->  
    typename std::result_of< Fn( A, B ) >::type  
{ return fn( pair.first, pair.second ); }
```

*// funkce pro rvalue*

```
template< typename T > // bere rvalue i lvalue  
void foo( T && );
```

*template< typename T > // bere jen rvalue*

```
void bar( typename  
    std::remove_reference<T>::type && );
```

# Přetížení funkce pro typy splňující vlastnost

princip *Substitution Failure Is Not An Error (SFINAE)*

- pokud při substituci typové proměnné v hlavičce funkce dojde k chybě (například nenalezení vnitřního typu ve třídě) nemusí to znamenat chybu kompilace
- příslušná funkce se nadále neuvažuje
- stále se mohou použít další přetížené verze

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

# Operátory: SFINAE

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

```
struct X : Eq {  
    int a, b;  
    bool operator==( const X &o ) const {  
        return a == o.a && b == o.b;  
    }  
};
```

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

# Operátory: SFINAE

- využívá toho, že operátory mohou být na úrovni namespace
- Eq poskytuje tag: zajišťuje, že instanciaci operátoru selže pro typy pro něž nebyla zamýšlena

## *Argument-Dependent Lookup (ADL)*

- Umožňuje hledat funkci kromě globálního a současného namespace také v namespace všech argumentů (a předků argumentů)
- podrobně, pro odvážné:  
<http://en.cppreference.com/w/cpp/language/adl>

# SFINAE v C++11 (složitější příklad)

```
#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); }
```

Mixin pro porovnávání pomocí SFINAE a ADL:

`http://cecko.eu/public/pb173/cviceni\_11`.

+ Rozšíření úlohy z minulého cvičení o práci s n-ticemi.