

# Hrátky s funkcemi

## PV173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

29. září 2014

Na co se podíváme?

- předávání funkcí do funkcí
- algoritmická knihovna C++
- základy funkcionálního programování v C++11

Co potřebujete

- C++11 kompatibilní kompilátor
- $\geq$  gcc 4.8, clang 3.4, VS CTP 2013
- možná budou fungovat i starší
- na školních pc module add gcc-4.8.2 (odeberte ostatní gcc moduly)

# Předávání funkce do funkce

Často se hodí

- různé callbacky
- porovnávací funkce pro sort
- funkce jako `for_each`, `accumulate`
- zavolání funkce pro každý vrchol stromu
- ...

# Předávání funkce do funkce – implementace

ukazatele na funkci – znáte z C

```
void qsort( void *ptr, size_t count, size_t size,  
           int (*comp)(const void *, const void *) )  
{ ... comp( a, b ) ... }
```

- + funguje v C++98 a bez šablon (a i v C)
- ± typová signatura callbacku přesně definovaná
- neumožňuje předat do funkce funktory (objekty implementující operátor ())
- nepodporuje všechny typy C++11 lambda
- lze předat NULL či `nullptr` místo funkce

# Předávání funkce do funkce – implementace

## C++11 function wrapper

```
#include <functional>
template< typename T >
void mysort( std::vector< T > &vec,
            std::function< bool(const T&, const T&) > cmp )
{ ... cmp( a, b ) ... }
```

- + funguje i s lambdami a funktory
- ± typová signatura callbacku explicitně definovaná, ale konverze možné
- zbytečný overhead, nelze inlinovat volání
- předaný objekt může být nevalidní → volání vyhodí `std::bad_function_call`
- jen C++11 a dále

# Předávání funkce do funkce – implementace

pomocí šablon

```
template< class RandomIt, class Compare >  
void sort( RandomIt first, RandomIt last,  
          Compare comp )  
{ ... comp( a, b ) ... }
```

- + funguje i s lambdami a funktory
- + funguje v C++98
- + kompilátor může snadno inlinovat callback
- + nelze jednoduše zavolat s NULL či **nullptr** (bez explicitní konverze na typ ukazatele na funkci)
- ± typová signatura callbacku definovaná jen jeho použitím
- pokud callback nelze zavolat s danými parametry je typová hláška hůře srozumitelná
- musí být šablona (tedy definovaný v .h)

# Předávání funkce do funkce – implementace

co teda?

- pokud je možné implementovat jako šablonu používejte poslední možnost
- šablonová verze se hojně (a výhradně) používá v C++ standardní knihovně
- pokud nutně musíte nemít šablonu, použijte `std::function`
- nepoužívejte ukazatele na funkce, jsou příliš omezující a nemají téměř žádné výhody proti `std::function`

# Algoritmická knihovna v C++

- <http://en.cppreference.com/w/cpp/algorithm>
- hlavičky `<algorithm>` a `<numeric>`
- řazení, vyhledávání, akumulace v kontejnerech; haldy...
- mnohé funkce berou funkci: `find_if`, `all_of`, `sort`, ...



# Příklad: suma pomocí `std::for_each`

Máme `std::vector< int >`, chceme spočítat sumu pomocí funkce `std::for_each`:

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first,
    InputIt last, UnaryFunction f );
```

**problém:** jak si držet sumu

- nesmíte použít globální ani statickou proměnnou

# Příklad: suma pomocí `std::for_each`

Máme `std::vector< int >`, chceme spočítat sumu pomocí funkce `std::for_each`:

```
template< class InputIt, class UnaryFunction >  
UnaryFunction for_each( InputIt first,  
    InputIt last, UnaryFunction f );
```

**problém:** jak si držet sumu

- nesmíte použít globální ani statickou proměnnou
- nelze udělat s pomocí běžné funkce, vyžaduje volatelný objekt

# Příklad: suma pomocí `std::for_each`

C++98 řešení: funktory (volatelné objekty)

```
int doSum( std::vector< int > &vec ) {  
    struct MkSum {  
        MkSum() : sum( 0 ) { }  
        int sum;  
        // uděláme objekt volatelným  
        void operator()( int val ) {  
            sum += val;  
        }  
    };  
    return std::for_each( vec.begin(), vec.end(),  
        MkSum() ).sum;  
}
```

... neprakticky dlouhé a nepřehledné

# Příklad: suma pomocí `std::for_each`

C++11 řešení: lambda funkce

```
int doSum( std::vector< int > &vec ) {  
    int sum = 0;  
    std::for_each( vec.begin(), vec.end(),  
        [&]( int val ) { sum += val; } );  
    return sum;  
}
```

- krátké a přehledné
- `[&]` na začátku lambda funkce říká, že chceme přistupovat ke všem lokálně přístupným proměnným pomocí reference
- kompilátor na pozadí vytvoří funktor

# Lambda funkce

```
[ <zachycení proměnných> ]( <parametry> )  
{ <tělo> }
```

- <parametry> jsou normální parametry funkce
- obdobně <tělo>
- <zachycení proměnných> dále
- návratová hodnota se v C++11 odvodí jen pokud je tělo tvaru **return** <výraz>, jinak je **void**
- v C++14 funguje odvození stejně jako pro **auto** funkce (= skoro vždy)
- lze specifikovat explicitně

```
[ <zachycení proměnných> ]( <parametry> )  
    -> <návratová hodnota>  
{ <tělo> }
```

# Lambda funkce

## zachytávání proměnných

Sekce <zachycení proměnných> může obsahovat:

- <proměnná> pro zachycení hodnotou
- &<proměnná> pro zachycení referencí
- = pro zachycení všech nespecifikovaných proměnných hodnotou
- & pro zachycení všech nespecifikovaných proměnných referencí
- může být prázdná (ale [] musí být)
- nezachycené proměnné jsou nedefinované a to včetně **this** pro lambdy vnořené v metodě objektu

```
[&sum,=]( int val ) { ... }
```

zachycuje `sum` referencí a ostatní použité proměnné hodnotou

# Lambda funkce

zachytávání proměnných – pozor

- zachycení proměnné nemění její životnost
- proměnné zachycené hodnotou jsou uvnitř lambdy **const**

Kde je chyba?

```
std::function< int( int ) > getAdder( int c ) {  
    return [&]( int val ) { return c + val; };  
}  
  
int main() {  
    auto add5 = getAdder( 5 );  
    std::cout << add5( 1 ) << std::endl;  
    return 0;  
}
```

# Cvičení I

Definujte funkci `forEach` chovající se jako knihovní `for_each` (funkce `f` může měnit hodnoty v kontejneru)

```
template< typename InputIt, /* ... */ >
void forEach( InputIt fst, InputIt lst,
             /* ... */ f );
```

Taky aby:

- a) používala ukazatel na funkci
- b) používala `std::function`
- c) používala šablonovaný callback

*poznámka:* typ hodnot je `typename InputIt::value_type`



# Cvičení II

Definujte funkci `forEach` chovající se jako knihovní `for_each` (funkce `f` může měnit hodnoty v kontejneru)

```
template< typename InputIt, /* ... */ >
void forEach( InputIt fst, InputIt lst,
             /* ... */ f );
```

Taky aby:

- a) používala ukazatel na funkci
- b) používala `std::function`
- c) používala šablonovaný callback

*poznámka:* typ hodnot je `typename InputIt::value_type`

Porovnejte jejich chování pro volání:

- `forEach(b,e, [](int &x) { std::cout << x; });`
- `forEach(b,e, [](int x) { std::cout << x; });`
- `forEach(b,e, [&](int &x) { sum +=x; });`

# Cvičení III

Definujte následující funkce bez použití cyklů a rekurze:

```
// spočítá průměr hodnot v rozsahu [fst, lst)
```

```
template< typename InputIt >
```

```
auto average( InputIt fst, InputIt lst )
```

```
    -> typename InputIt::value_type;
```

```
// spočítá průměr těch hodnot v rozsahu
```

```
// [fst, lst), které splňují pred
```

```
template< typename InputIt, typename UnaryPred >
```

```
auto average_if( InputIt fst, InputIt lst,
```

```
    UnaryPred pred )
```

```
    -> typename InputIt::value_type;
```

Bonus: definujte average pomocí average\_if.

# Typové chyby – ukazatel na funkci

```
1  int callFP( int(*fp)(int) ) { return fp(5); }
2  int main() {
3      callFP( []( int x ) { return x + 1; } );
4      callFP( []( const int &x ) { return x; } );
5      callFP( []( int &x ) { return x + 2; } );
6  } /*
7  ferr_fp.cpp:4:5: error: no matching function for
    call to 'callFP'
8      callFP( []( const int &x ) { return x; } );
9      ^~~~~~
10 ferr_fp.cpp:1:5: note: candidate function not
    viable: no known conversion from '<lambda at
    ferr_fp.cpp:4:13>' to 'int (*)(int)' for 1st
    argument
11 int callFP( int(*fp)(int) ) { return fp(5); }
12     ^
```

... totéž pro řádek 5

# Typové chyby – std::function

```
1  #include <functional>
2  void call(std::function<int(int)> f) { f(5); }
3  int main() {
4      call( []( int x ) { return x + 1; } );
5      call( []( const int &x ) { return x; } );
6      call( []( int &x ) { return x + 2; } );
7  } /*
8  ferr_fw.cpp:6:5: error: no matching function for
    call to 'call'
9      call( []( int &x ) { return x + 2; } );
10     ^~~~
11  ferr_fw.cpp:2:6: note: candidate function not
    viable: no known conversion from '<lambda at
    ferr_fw.cpp:6:11>' to 'std::function<int
    (int)>' for 1st argument
12  void call(std::function<int(int)> f) { f(5); }
13      ^
14  1 error generated.
15  */
```

# Typové chyby – šablony

```
1  template< typename F >
2  void call( F f ) { f(5); }
3  int main() {
4      call( []( int x ) { return x + 1; } );
5      call( []( const int &x ) { return x; } );
6      call( []( int &x ) { return x + 2; } );
7  }
```

# Typové chyby – šablony

```
ferr_t.cpp:2:20: error: no matching function for
      call to object of type '<lambda at
      ferr_t.cpp:6:11>'
void call( F f ) { f(5); }
```

```
^
ferr_t.cpp:6:5: note: in instantiation of
      function template specialization
      'call<<lambda at ferr_t.cpp:6:11> >'
      requested here
      call( []( int &x ) { return x + 2; } );
```

```
^
ferr_t.cpp:6:11: note: candidate function not
      viable: expects an l-value for 1st argument
      call( []( int &x ) { return x + 2; } );
```

```
^
ferr_t.cpp:6:11: note: conversion candidate of
      type 'int (*)(int &)'
1 error generated.
```

# Typové chyby – závěr

- funkční ukazatel je příliš restriktivní
- C++11 ve skutečnosti negarantuje, že příklad na `std::function` se nezkompiluje!
- může se zkompilevat a volání vyhodit `std::bad_function_call`
- C++14 již chybovou hlášku garantuje
- u šablon se chyba generuje v místě volání callbacku
- musíme procházet poznámky a zjišťovat co se vlastně stalo

# Lambda funkce v C++14

Odvozování návratového typu (pokud není uveden)

- všechny **return** výrazy musí mít stejný typ

Generické lambda funkce

- `[] ( auto x, auto y ) { return x < y; }`
- chová se jako kdyby každý `auto` parametr byl samostatně šablonovaný, tedy jako

```
struct Closure {  
    template< typename T1, typename T2 >  
    auto operator()( T1 x, T1 y )  
    { return x < y; }  
};
```

Překlad: gcc/clang s `-std=c++1y` (`-std=c++14` od clang 3.5).



# Domácí úkol

Implementujte třídu `Vector< T >`, která se chová stejně jako `std::vector< T >` a navíc má dotazovací funkce (`accumulate`, `sum`, `average`, `median`, `forEach`, `map`, ...) inspirované funkcionálním programováním a C# LINQ (které je samo inspirované funkcionálním programováním).

Kompletní zadání naleznete na [http://cecko.eu/public/pb173/cviceni\\_03](http://cecko.eu/public/pb173/cviceni_03).

V ISu je odevzdávárna. Odevzdávejte do 13. 10. 2014 14:00.