

FUNKČNÍ OBJEKTY, LAMBDY; TYPOVÉ KONVERZE

PB161 PROGRAMOVÁNÍ V JAZYCE C++

Nikola Beneš

4. dubna 2023

Ukazatele na funkce

- drží adresu (volné) funkce
- klasická deklarace poněkud nečitelná
 - alternativy: typové aliasy, **auto**¹

```
void (*ptr1)(int) = foo;
```

```
using fun = void(int);
```

```
using fun_ptr = fun*;
```

```
fun_ptr ptr2 = foo;
```

```
auto* ptr3 = foo;
```

¹dedukce s **auto** se nedá použít pro generické / přetížené funkce

Neadresovatelné funkce

- funkce, na něž nesmíme mít ukazatel / referenci
 - (nedefinované chování)
- `main`
- všechny funkce ze standardní knihovny, pokud nejsou explicitně označené jako *addressable*
 - v současnosti jen I/O manipulátory (uvidíme později)
 - https://en.cppreference.com/w/cpp/language/extending_std#Addressing_restriction

Typové konverze (implicitní)

- funkce ↔ ukazatel na funkci
 - → při typové dedukci **auto** (tzv. *decay*)
 - ← při volání
- **nullptr** → ukazatel na funkci
- ukazatel na **noexcept** funkci → ukazatel bez **noexcept**
- žádné jiné, ani explicitní
 - zejména nemůžeme standardně konvertovat na **void*** (může existovat platformově závislá konverze)

- funkce, která má jako parametr jinou funkci

```
using int_pred = bool(int);
```

```
std::vector<int> filter(const std::vector<int>& vec,  
                      int_pred* pred) {  
    std::vector<int> result;  
    for (int x : vec)  
        if (pred(x))  
            result.push_back(x);  
    return result;  
}
```

- chceme generickou funkci **filter**, která bude fungovat pro libovolné kontejnery a libovolné predikáty
- jaký bude typ proměnné **result**?
 - potřebujeme získat typ prvků uvnitř **cont**
- první nápad: **decltype** můžeme použít s libovolným výrazem
 - výraz se nevyhodnotí, jen se otypuje
 - **decltype(*cont.begin())** je (konstantní) reference na prvek kontejneru **cont**
- potřebovali bychom z typu „odstranit **const** a **&**“

Část knihovny `<type_traits>`

- různé manipulace s typy pro metaprogramování
- mimo záběr předmětu, ale jedna transformace se zde hodí:

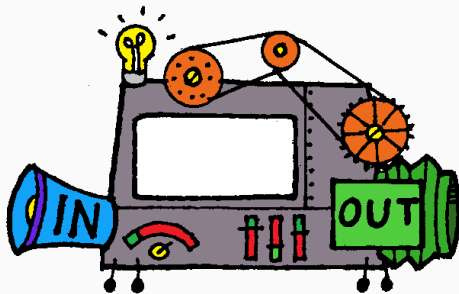
`std::decay_t< typ >`

- typ, který by se dedukoval při předávání hodnotou
 - `auto` bez `const` nebo `&`
- pro běžné typy: odstraní `const` a reference
- pro funkce: funkční ukazatel
- (pro Cíčková pole: ukazatel na první prvek)
- `std::decay_t<decltype(x)> value;` je tedy *jakoby*:
 - `auto copy = x; decltype(copy) value;`
 - ale bez zbytečného vytváření kopie

```
auto filter(const auto& cont, auto pred) {  
    using T = std::decay_t<decltype(*cont.begin())>;  
    std::vector<T> result;  
    for (auto& x : cont)  
        if (pred(x))  
            result.push_back(x);  
    return result;  
}
```

- *všimněte si:* parametru `pred` nemůžeme předat generickou / přetíženou funkci

FUNKČNÍ OBJEKTY



Přetížený **operator()**

- vždy jako metoda; libovolný počet parametrů
- umožňuje volat objekty pomocí funkčního volání

```
class adder { // not the snake
    int val;

public:
    adder(int v) : val(v) {}
    int operator()(int x) const { return x + val; }
};
```

Použití generické funkce s funkčním objektem

```
struct even {
    bool operator()(auto num) const {
        return num % 2 == 0;
    }
};

struct less_than {
    int value;
    bool operator()(int num) const {
        return num < value;
    }
};

auto v1 = filter(v, even{});
auto v2 = filter(v, less_than{5});
```

`std::plus`, `std::multiplies`, ...

- funkční objekty obalující
 - aritmetické operace
 - porovnání
 - logické a bitové operace
- pro použití s generickým kódem
 - (zejména s knihovnou algoritmů – uvidíme později)

LAMBDA-VÝRAZY



Lambda expressions—lambdas—are a game changer in C++ programming. That's somewhat surprising, because they bring no new expressive power to the language. Everything a lambda can do is something you can do by hand with a bit more typing. But lambdas are such a convenient way to create function objects, the impact on day-to-day C++ software development is enormous.

(Scott Meyers, Effective Modern C++)

Anonymní funkce

Lexikální uzávěry (*closures*)

- možná znáte z Pythonu nebo jiných moderních jazyků
- (nepojmenované) *funkční objekty* definované uvnitř výrazů
- mohou zachytávat kontext
 - lokální proměnné z místa definice
- typické použití:
 - lokální funkční objekty
 - s generickým kódem (algoritmy)

```
auto v1 = filter(v,  
    [](int num) { return num % 2 == 0; });
```

[chycený kontext](parametry) -> návratový typ { tělo }

- -> návratový typ se může vynechat
 - dedukce z **return**
 - jako **auto** návratový typ u funkcí
- parametry a tělo jako u funkcí
 - generické parametry pomocí **auto**
 - prázdný seznam parametrů () možno vynechat
- chycený kontext (*capture*) umožňuje uložit v objektu lambdy hodnoty (lokálních) proměnných z místa, kde je vytvořena

Typ lambda

- speciální, programátorem nepojmenovatelný
 - je třeba použít **auto**
 - nebo **std::function** (uvidíme příště)
 - (nebo šablony – mimo záběr předmětu)
- lambda s prázdným kontextem se dá implicitně konvertovat na funkční ukazatel

Objekt lambda

- funkční objekt
- životní cyklus jako jakýkoli jiný objekt
- chycený kontext – položky objektu

- `[]` nic
- `[=]` všechny volné proměnné v těle lambdy *hodnotou*
- `[&]` všechny volné proměnné v těle lambdy *referencí*
- `[x]` proměnnou `x` hodnotou
- `[&x]` proměnnou `x` referencí
- `[x, &y]` proměnnou `x` hodnotou, proměnnou `y` referencí
- `[=, &x]` proměnnou `x` referencí, ostatní volné proměnné v těle lambdy hodnotou
- `[x = 42]` uvnitř lambdy je nová položka s hodnotou 42 (deklarace jako pomocí **auto**)
- `[x = std::move(x)]` přesun objektu dovnitř lambdy
- `[&x = value]` zachycení proměnné `value` referencí s přejmenováním na `x`

<https://en.cppreference.com/w/cpp/language/lambda>

```
auto less_than(auto value) {  
    return [=](auto num) { return num < value; };  
}
```

- funkční objekt lambdy obsahuje položku `value`
- při inicializaci lambdy se do položky `value` *zkopíruje* hodnota lokální proměnné (parametru) `value`
- použití `[&]` by zde byla chyba, proč?
 - položka `value` v lambdě by byla referenčního typu
 - lokální proměnná `value` zanikne při opuštění funkce
 - *dangling reference*

Vlastnictví

- položky zachycené hodnotou (`[=]`, `[x]`, `[x = ...]`)
 - jsou vlastněny objektem lambdy
 - pro `[=]`, `[x]` kopie vnějších objektů, u `[x = ...]` dle inicializace
 - jejich život zaniká po skončení života lambdy
- položky zachycené referencí (`[&]`, `[&x]`, `[&x = ...]`)
 - jsou pouze reference
 - objekt lambdy odkazované objekty nevlastní
 - musíme zaručit, že odkazované objekty lambdu přežijí

```
[a, &b, c = 1, &d = num](auto x) { tělo }
```

- typ této lambdy vypadá nějak takto:

```
class tajné_jméno {  
    typ_a a;  
    typ_b& b;  
    int c;  
    typ_num& d;  
public:  
    auto operator()(auto x) const { tělo }  
};
```

- **a** se inicializuje kopií lokální proměnné **a**
- **b** se inicializuje referencí na lokální proměnnou **b**
- **c** se inicializuje hodnotou **1**
- **d** se inicializuje referencí na lokální proměnnou **num**

operator() lambda

- implicitně **const**
- důsledek: položky zachycené hodnotou nesmíme modifikovat
- (položky zachycené referencí ano, **const** je plytké)
- chceme-li nekonstantní operátor, použijeme **mutable**:

```
[kontext](parametry) mutable { ... }2
```

```
auto v3 = filter(v,  
    [i = 0](int num) mutable {  
        return num == i++;  
    });
```

²prázdný seznam parametrů v tomto případě nemůžeme vynechat (do C++23)

[this]

- v lambdě definované uvnitř metody
- zachytí **this** hodnotou, tedy *aktuální objekt jakoby referencí*
 - musíme zaručit, že aktuální objekt lambda přežije
- uvnitř lambdy můžeme přistupovat k položkám / metodám aktuálního objektu
 - bez nutnosti psaní **this**
 - jako bychom byli uvnitř metody
- **this** se implicitně zachytí též s [&]

[*this]

- zachytí aktuální objekt *hodnotou*
- tj. uvnitř lambdy bude uložena jeho *kopie*

```
struct node {
    using ptr = std::unique_ptr<node>;
    int value;
    ptr left, right;
    void preorder(auto& fun) {
        fun(value);
        if (left) left->preorder(fun);
        if (right) right->preorder(fun);
    }
};

struct tree {
    node::ptr root;
    void preorder_visit(auto fun) {
        if (root) root->preorder(fun);
    }
};
```


TYPOVÉ KONVERZE

Konverze pomocí konstruktoru

- směr *cizí typ* → *náš typ*
- libovolný konstruktor volatelný s *jedním parametrem*
 - (pamatujte na implicitní parametry)
- implicitní
 - (dá se zakázat pomocí **explicit** – mimo záběr předmětu)

```

class fraction {
    int num, denom;

public:
    fraction(int num = 0, int denom = 1)
        : num(num), denom(denom) { normalize(); }

    // ...

};
  
```

Konverze pomocí operátoru

- směr *náš typ* → *cizí typ*
- **operator** typ
- musí být metoda, bez parametrů, *bez návratové hodnoty*
- implicitní
 - (i zde můžeme použít **explicit**)

```

class fraction {
    // ...

    operator double() const {
        return static_cast<double>(num) / denom;
    }

    // ...

};
  
```