Funkční objekty, lambdy; typové konverze

PB161 Programování v jazyce C++

Nikola Beneš

9. dubna 2024

Funkční ukazatele

fun_ptr.cpp

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;
```

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

Funkční ukazatele

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

Funkční ukazatele

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 vyšších řádů

filter1.cpp

- 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;
}
```

Generické funkce vyšších řádů

filter2.cpp

```
auto filter(const auto& cont, auto pred) {
```

- 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 `&`“

Odbočka: Transformace typů

Část knihovny `<type_traits>`

- různé manipulace s typy pro metaprogramování
- mimo záběr předmětu, ale jedna transormace 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 Céčková 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

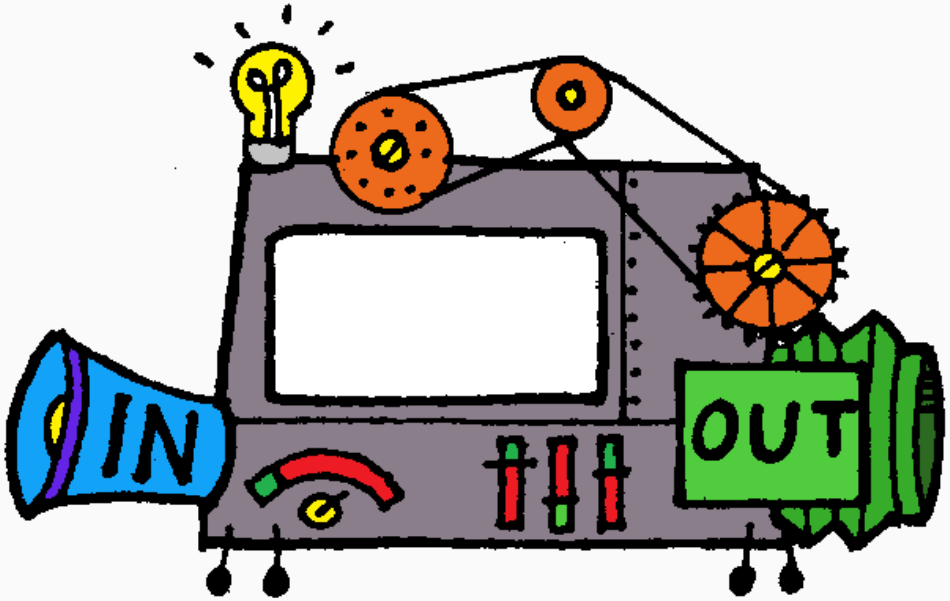
Generické funkce vyšších řádů

filter2.cpp

```
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



Funkční objekty

op_fun_obj.cpp

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; }
};
```

Funkční objekty

filter3.cpp

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});
```

Funkční objekty ve standardní knihovně

`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++)

filter4.cpp

~~Anonymní funkce~~

Lexikální uzávěry (*closures*)

- možná znáte z Pythonu nebo jiných moderních jazyků
- (nepojmenované) *funkční objekty* definované uvnitř výrazů
- můžou 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; });
```

Anatomie lambda-výrazu

```
[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

Lambda-výrazy

Typ lambdy

- 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 lambdy

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

Zachytávání kontextu

- `[]` 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

Lambda se zachytáváním

filter4.cpp

```
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*

Chycený kontext

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

Chycený kontext

```
[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`

Operátor volání

filter4.cpp

`operator()` lambdy

- 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++;
    });
```

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

Zachycení aktuálního objektu

this_capture.cpp

`[this]`

- v lambdě definované uvnitř metody
- zachytí **`this`** hodnotou, tedy *aktuální objekt jakoby referencí*
  - musíme zaručit, že aktuální objekt lambdu 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*

Příklad: Procházení stromu

tree.cpp

```
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

convert.cpp

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(); }

    // ...
};
```

convert.cpp

Konverze pomocí operátoru

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

```
class fraction {
    // ...

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

    // ...
};
```

Bonus: IIFE

`iife.cpp`

Immediately-Invoked Function Expression (IIFE)

- lambda, kterou okamžitě zavoláme
- použití jako „blok, který má hodnotu“
  - tj. blok na místě výrazu
  - (možná znáte z jiných jazyků)
  - výhoda: omezená existence lokálních proměnných
- použití pro složitější inicializaci
  - pro `const` proměnné, reference, …