

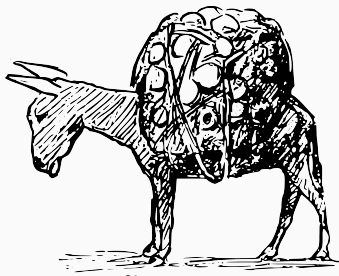
METODY A OPERÁTORY

PB161 PROGRAMOVÁNÍ V JAZYCE C++

Nikola Beneš

28. února 2023

PŘETĚŽOVÁNÍ FUNKCÍ



Deklarace funkce

- „hlavička“ – návratový typ, typy a případně jména parametrů
- nenásleduje-li tělo, jen ;, jde o dopřednou deklaraci
 - v hlavičkových souborech, pro nepřímou rekurzi apod.

Definice funkce

- deklarace + tělo
- tělo je typicky blok, ale může mít i speciální podobu
 - (uvidíme později)

Deklarace a definice typu

- dopředná deklarace **struct** jméno;
 - není možné pracovat s *hodnotami* typu jméno (v tělech funkcí) před skutečnou definicí

- různé funkce se stejným jménem, ale různými definicemi
- musí se lišit typem nebo počtem parametrů
- nestačí, když se liší pouze návratovým typem!

```
void f(int x) {  
    std::cout << "int: " << x << '\n';  
}  
void f() {  
    std::cout << "no parameters\n";  
}  
void f(double d) {  
    std::cout << "double: " << d << '\n';  
}  
  
f(42); f(3.14); f();
```

PŘETĚŽOVÁNÍ (OVERLOADING) FUNKCÍ

Pravidla pro přetěžování

- přesná shoda
- implicitní typové konverze
 - standardní
 - uživatelsky definované (uvidíme později)
 - přednost má *promotion*¹ (konverze na typ s větším rozsahem)
- jsou-li dvě možnosti stejně „daleko“, jde o chybu
 - nejednoznačné (*ambiguous*) volání

```
void f(int);
```

```
void f(double);
```

```
f(3.14f); // floating-point *promotion*, OK
```

```
f(42u); // two possible *conversions*, error
```

¹viz https://en.cppreference.com/w/cpp/language/implicit_conversion

Nekonstantní reference

- má přednost před konstantní referencí
- použije se, pokud je argumentem modifikovatelná *l-hodnota*
 - proměnná apod.
 - výraz, který je typu nekonstantní reference

```
void f(int&) {  
    std::cout << "reference\n";  
}
```

```
void f(const int&) {  
    std::cout << "const reference\n";  
}
```

Poznámka: Přetížení funkce pro referenci a hodnotu vede k nejednoznačnosti.

Funkce s parametry `auto`

- použije se, nenajde-li se funkce s přesnou shodou
 - (hodnotou nebo referencí)
- tj. předtím, než se začnou zkoušet implicitní konverze
 - nedojde ani na *promotion*

```
void f(auto) {}
```

```
void f(int x) {  
    std::cout << "int: " << x << '\n';  
}
```

```
f(3);    // only this one calls f(int)  
f('x');  
f(0.0);
```

Parametry s implicitní hodnotou

- za povinnými parametry (tj. co nejvíce „vpravo“)
- jako inicializace u deklarace proměnné
- hodnota vzniká **v okamžiku volání funkce**
- více deklarácí: inicializace jen u té první
 - zvyk pro čitelnost: opakovat v komentáři
- rozumně *nefunguje* pro generické parametry (**auto**)

```
void f(int x, int y = 7, std::vector<int> v = {}) {  
    // ...  
}
```


Nekvalifikované jméno (nemá před sebou ::) – *velmi zjednodušeně*

- postupuje se od vnitřního rozsahu směrem k vnějším
 - vnitřní blok uvnitř funkce
 - vnější blok atd.
 - položka složeného typu – uvnitř metody (uvidíme za chvíli)
 - jmenný prostor, v němž je funkce deklarovaná
 - vnější jmenný prostor atd.
 - jmenné prostory dle **using namespace**
- u funkcí navíc tzv. *argument-dependent lookup*

Kvalifikované jméno (má před sebou ::)

- hledá se uvnitř jmenného prostoru nebo složeného typu
- :: jméno se hledá v globálním jmenném prostoru

Detaily: <https://en.cppreference.com/w/cpp/language/lookup>
(velice obtížné čtení)

Hledání podle argumentů (*Argument-dependent lookup, ADL*)

- kromě klasického hledání se hledá i uvnitř jmenných prostorů *souvisejících s argumenty*
- díky tomuto funguje přetěžování některých operátorů

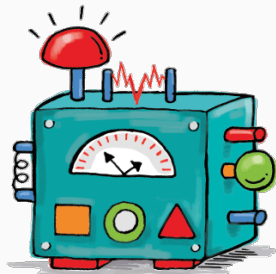
```
std::vector<int> v;  
erase(v, 0); // ADL finds std::erase
```

- má přednost před deklarací **using**
- idiom pro použití funkce **swap**
 - od C++20 možno nahradit **std::ranges::swap(a, b);**

```
using std::swap;  
swap(a, b);
```

- pokud typ **a** nebo **b** je ve jmenném prostoru, kde je definována funkce **swap**, použije se ta; jinak se použije **std::swap**

METODY



Objekt (dle standardu C++)

- úložiště pro hodnotu, s identitou
 - má velikost, životnost² (*lifetime*), typ, ... (viz odkaz níže)
- abstrakce místa v paměti
 - místo v paměti uchovává bajty, má adresu
 - objekt uchovává hodnoty svého typu, má identitu
- objekty se často realizují jako sekvence bajtů v paměti, ale nemusí tomu tak nutně být

Dočasné objekty

- dočasné hodnoty se za určitých okolností stanou objekty
- „když se na ně někdo podívá“

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

²bude tématem příští přednášky

Členské proměnné (*member variables*)

- položky složeného datového typu **struct**
- v jiných jazycích „atributy“
 - (v C++ má tento pojem jiný význam)

Členské funkce (*member functions*)

- funkce deklarované uvnitř **struct**
- operují nad hodnotami daného typu
- jinde nazývané „metody“
 - budeme jim tak říkat i zde
- ne-členským funkcím (*non-member function*)
někdy říkáme „volné“ (*free*)

Deklarace a definice metod

- deklarace uvnitř definice typu
- definice:
 - uvnitř definice typu (vhodné pro malé metody)
 - odděleně (větší metody)

```
struct canvas {  
    std::map<std::tuple<int, int>, char> pixels;  
    // ...  
    void clear() { pixels.clear(); }  
  
    void draw() const;  
    // ...  
};
```

Tělo metody

- jako tělo funkce
- přístup k položkám aktuálního objektu – přímo jejich jmény
- volání jiných metod aktuálního objektu – přímo jejich jmény
 - nezávisí na pořadí deklarace
- ***this** je reference na aktuální objekt (více o **this** později)

Volání metody

- pomocí tečkové notace
- výraz před tečkou – aktuální objekt

```
canvas c;  
c.pixel(1, 1) = 'X';  
c.pixel(2, 3) = 'Y';  
c.draw();
```

Kvalifikátor `const`

- součást hlavičky metody, píše se za parametry
- metoda nemění hodnotu objektu – hlídá překladač
- máme-li konstantní (referenci na) objekt, smíme volat pouze `const` metody
- možno mít dvě metody stejného jména, s `const` a bez

Doporučení

- deklarujte jako `const` všechny metody, které nemají měnit stav objektu
- ochrana proti chybám

Metody pro volání `object.metoda1().metoda2().metoda3()`

- někdy oblíbený³ idiom pro některé návrhové vzory
 - (*named parameter, builder pattern, fluent interface, ...*)
- implementace:
 - návratová hodnota je *reference* na aktuální typ
 - každé volání vrátí *referenci* na aktuální objekt
 - pomocí **return *this;**
- používá se u některých operátorů (uvidíme za chvíli)

³existují argumenty pro i proti; závisí na konkrétním použití



Klíčové slovo **friend**

- deklarace *volné* funkce (případně i s definicí), která úzce souvisí s naším typem – uvnitř definice typu
 - tj. není to metoda, volá se jako běžná volná funkce
- nechceme nebo nemůžeme-li implementovat jako metodu
- časté použití u operátorů (za chvíli)

- technicky má dva významy:
 - **friend** smí přistupovat k soukromým členům (uvidíme později)
 - **friend** funkce včetně definice uvnitř typu nejsou vidět pro normální hledání jména, ale najde je ADL (tzv. *hidden friend idiom*, má různé výhody, mimo záběr předmětu)
- existují i **friend** typy (opět uvidíme později)

PŘETĚŽOVÁNÍ OPERÁTORŮ



Přetížené operátory (*overloaded operators*)

- už jste viděli
- téměř v libovolném jazyce
 - např. aritmetické operátory pro `int` vs. `float`
 - `+` pro zřetězení, spojení seznamů (Python) apod.
- C++ umožňuje přetížit operátory pro vlastní datové typy

Proč používat?

- lepší čitelnost kódu, snazší použití uživatelských typů
- (redukce chyb)

```
BigInteger a, b, c;  
c += a * b;  
// vs  
c = c.add(a.multiply(b));
```

Syntax

- volná funkce nebo metoda
 - v případě metody je prvním operandem aktuální objekt
- **operator** a označení operátoru

Zdroje

- <https://en.cppreference.com/w/cpp/language/operators>
- https://en.wikipedia.org/wiki/Operators_in_C_and_C++
- <https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading>
- <https://isocpp.org/wiki/faq/operator-overloading>

Operator overloading makes life easier for the users of a class, not for the developer of the class! (C++ FAQ)

Operátor jako volná funkce

- unární operátor – jeden parametr
- binární operátor – dva parametry
- často jako **friend**

$$a + b \implies \text{operator}+(a, b)$$

Operátor jako metoda

- první (levý) operand je aktuální objekt
- počet parametrů: o jeden méně než pro volnou funkci

$$a + b \implies a.\text{operator}+(b)$$

Kterou formu zvolit?

- některé operátory není možno přetížit
 - ::, ., .*, ? :
- některé musí být metody
 - =, [], ->, ()
- ostatní je možno implementovat oběma způsoby

Doporučení

- není-li typ prvního operandu „náš“: volná (**friend**) funkce
- unární operátor: metoda
- binární operátor
 - chová se k operandům *stejně*: volná (**friend**) funkce
 - chová se k prvnímu operandu jinak: metoda

Omezení

- nelze definovat nové operátory
- nelze měnit počet parametrů (kromě operátoru ())
- nelze definovat operátory pro vestavěné typy
 - typ alespoň jednoho operandu musí být uživatelsky definovaný
- nelze měnit prioritu ani asociativitu

Binární **operator**+, **operator**+= apod.

- doporučeno: vždy obě verze (samostatná, s přiřazením)
- +: volná funkce, vrací hodnotu
 - neměla by modifikovat operandy
- +=: metoda, vrací referenci na aktuální objekt (***this**)
 - neměla by modifikovat pravý operand
- často je výhodné implementovat + pomocí += (viz další slajd)
 - složité objekty s „drahým“ kopírováním
- někdy je ale výhodná obrácená implementace (+= pomocí +)
nebo dvě oddělené – závisí na konkrétní situaci
- interakce s jinými typy: pamatujte na symetrii
 - např. násobení vektoru skalárem zleva i zprava

```
struct counter {
    std::map<char, int> freq;
    // ...

    counter& operator+=(const counter& other) {
        // ...
        return *this;
    }

    friend counter operator+(counter lhs,
                             const counter& rhs) {
        lhs += rhs;
        return lhs;
    }
};
```

```
friend counter operator+(counter lhs,  
                          const counter& rhs) {  
    lhs += rhs;  
    return lhs;  
}
```

- levý operand je *hodnota*, ne reference
 - je-li skutečný argument *l-hodnota*, vytvoří se kopie
 - je-li skutečný argument dočasná hodnota, kopie se nevytvoří
- hodnotový parametr za return umožňuje tzv. *přesun*
- vhodné pro sekvence typu $a + b + c + d$ (levá asociativita)

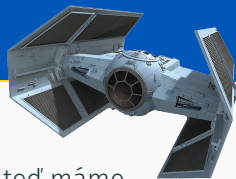
více o tom, kdy se dějí a nedějí kopie, a co znamená přesun – příště

operator++, operator--

- typicky jako metody
- postfixová varianta má navíc (nepoužitý) parametr typu `int`
- prefix: vrací referenci na aktuální objekt
- postfix: vrací hodnotu (*kopii* objektu před modifikací)

```
label& operator++() { // prefix
    // ... actual code ...
    return *this;
}

label operator++(int) { // postfix
    auto copy = *this;
    ++*this;
    return copy;
}
```



operator==, **operator**< a další

- před C++20 bylo třeba přetížit každý zvlášť... ..ale teď máme

operator<=>

- vrací speciální hodnotu, kterou je možné
 - porovnávat s literálem `0` pomocí `==`, `<` atd., přičemž $(a <= b) < 0$ znamená $a < b$ apod.
 - použít s funkcemi `std::is_eq`, `std::is_lt` atd.
- návratový typ je
 - `std::strong_ordering` – totální⁴ uspořádání, rovnost znamená nerozlišitelnost
 - `std::weak_ordering` – totální předuspořádání (povoluje rovnost různých hodnot)
 - `std::partial_ordering` – částečné předuspořádání (povoluje neporovnatelné hodnoty, rovnost různých hodnot)

⁴tj. lineární; pro každé a, b platí jedno z $a < b$, $a == b$, $a > b$

Automatické generování všeho

```
friend auto operator<=>(const full&,
                        const full&) = default;
```

- vygeneruje jak <=>, tak všechny ostatní porovnávací operátory
- vygenerované ==, != fungují po složkách
- ostatní lexikograficky v pořadí deklarace (jako u `std::tuple`)
- návratový typ <=> dle „nejslabší“ položky

Automatické generování jen (ne)rovnosti

```
friend bool operator==(const only_eq&,
                      const only_eq&) = default;
```

- vygeneruje == a !=
- nechceme-li ostatní (nebo nedávají smysl)

Uživatelsky definované chování

- implementujeme-li ==, automaticky se doplní !=
- implementujeme-li <=>, automaticky se doplní <, >, <=, >=
 - ale ne operátory (ne)rovnosti
 - důvod: rovnost typicky umíme implementovat efektivněji (zejména pro případ, že se hodnoty nerovnají)

```
friend auto operator<=>(const rev_lex& a,  
                        const rev_lex& b) {  
    if (auto cmp = a.y <=> b.y; cmp != 0) {  
        return cmp;  
    }  
    return a.x <=> b.x;  
}
```

Unární aritmetické operátory (unární +, -, ~)

- typicky jako metoda vracující novou hodnotu
- `my_type operator-() const { /* ... */ }`

Přiřazení =

- uvidíme příště

Dereference (unární *, ->)

Funkční volání (operátor ())

Typové konverze

- uvidíme na konci druhého bloku

Proudový vstup / výstup >>, <<

- uvidíme na konci třetího bloku

Operátor indexování []

- necháme na navazující předmět
- (vhodná implementace potřebuje šablony, `static`, ...)

Vynecháme úplně

- operátor `->*`

Nedoporučeno přetěžovat

- `&&`, `||`, `,` (operátor čárka), unární `&`

Doporučení

- nezapomeňte na princip nejmenšího překvapení
- používejte operátory jen tam, kde je jasný jejich význam
- implementujte operátory tak, aby měly očekávanou sémantiku
 - které operátory objekt modifikují
 - které operátory vytvářejí nové hodnoty
 - co se čeká, že vracejí
- (někdy porušuje i standardní knihovna...)

I saw cout being shifted "Hello world!" times to the left and stopped right there. (Steve Gonedes)