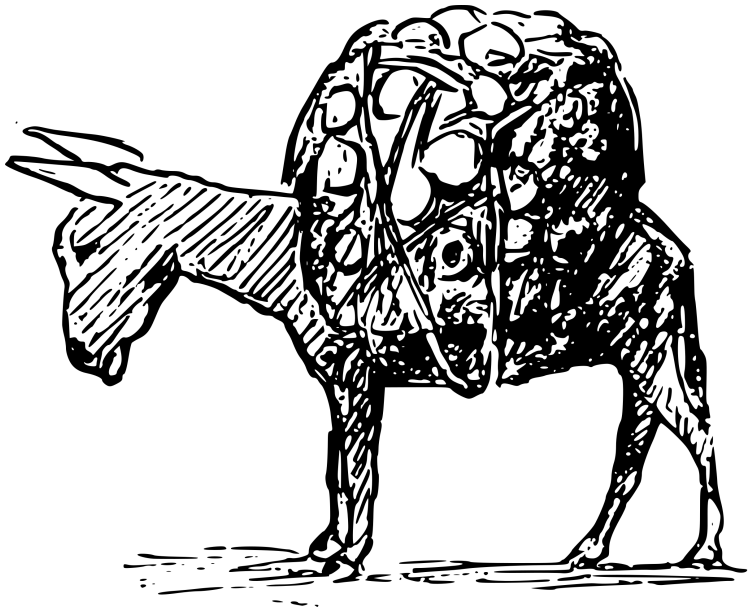
PB161 Programování v jazyce C++
Přednáška 4

Přetěžování funkcí
Konstruktory a destruktory

Nikola Beneš

11. října 2016

Přetěžování funkcí



Přetěžování (overloading)

- různé funkce/metody se stejným jménem, ale různou definicí
- musí se lišit typem nebo počtem parametrů nebo `const` (u metod)

```
void f(int x) {
    cout << "f s parametrem int: " << x << endl;
}
void f() {
    cout << "f bez parametrů" << endl;
}
void f(double d) {
    cout << "f s parametrem double: " << d << endl;
}

f(3);   // zavolá void f(int)
f(5.0); // zavolí void f(double)
f();    // zavolá void f()
```

- Pozor! V C++ nestačí, pokud se liší pouze návratovým typem; proč?

Přetěžování (overloading) – pokr.

Pravidla pro přetěžování

- přesná shoda
- rozšíření typu
- typové konverze, ...

```
class File {
// ...
public:
    void write(int num);
    void write(std::string str);
// ...
};

File f;
// ...
f.write("Hello"); // volá File::write(std::string)
f.write(0.0);     // volá File::write(int);
```

Přetěžování a reference

- nekonstantní reference má přednost

```
void f(int &);
void f(const int &);

int x;

f(x); // zavolá se první funkce
f(5); // zavolá se druhá funkce

int & ref = x;
const int & cref = x;
f(ref);  // první
f(cref); // druhá
```

- přetěžování pro referenci a hodnotu – nejednoznačnost (ambiguity)

Přetěžování a NULL

- v C++03: `NULL` je definováno jako `0`

```
void f(int x);
void f(char * s);

f(NULL); // zavolá se PRVNÍ funkce! (nebo chyba kompilátoru)
```

- proto máme v C++11 speciální ukazatel **`nullptr`**

```
f(nullptr); // OK, zavolá se druhá funkce
```

- `NULL` může být v C++11 definováno různě, proto je lépe jej nepoužívat a zvyknout si na používání **`nullptr`**

Přetěžování (overloading)

Pro zvídavé: Jak to ve skutečnosti funguje?

- překladač C++ mění jména funkcí, přidává k nim typy parametrů
  - tzv. *name mangling*
  - není žádný standard, závisí na konkrétním překladači
- příklad (gcc, clang):

`void fun(int, char)` → `_Z3funic`

`int fun(int&)` → `_Z3funRi`

Implicitní parametry

Parametry s implicitní hodnotou

- funkce, metody, konstruktory, ...
- musí být nejvíce vpravo v seznamu parametrů

```
void f(int x, int y = 10, int z = 20);

f(3, 4, 5); // zavolá se f(3,  4,  5)
f(3, 4);    // zavolá se f(3,  4, 20)
f(3);       // zavolá se f(3, 10, 20)

void g(int x = 10, int y); // chyba!
```

Implicitní parametry (pokr.)

Oddělení deklarace a definice

```
// soubor.h

void f(int x = 10, int y = 20);

// soubor.cpp

void f(int x /* = 10 */, int y /* = 20 */) {
    // ...
}
```

Konstruktory a destruktory

Konstruktory

Už víte:

- konstruktor je speciální metoda volaná při inicializaci objektu
- konstruktor má tzv. inicializační sekci
- jméno konstruktoru = jméno třídy

Možná nevíte:

- konstruktor není vždy nutné psát, vygeneruje se defaultní

Přetěžování konstruktorů

- jako přetěžování funkcí/metod
- (od C++11) konstruktory můžou v inicializační sekci volat jiné konstruktory

[ukázka]

Konstruktory (pokr.)

Pořadí volání konstruktorů

- konstruktory atributů se volají před konstruktorem objektu
  - ve skutečnosti se volají v rámci inicializační sekce
- konstruktory se volají v pořadí deklarací ve třídě
  (pozor! ne v pořadí daném inicializační sekcí)
  - varování kompilátoru `-Wreorder`
- volání konstruktorů při dědičnosti (později, v přednášce o OOP)

[ukázka]

Kdy se volá konstruktor

- lokální objekty (na zásobníku): v místě deklarace
- globální objekty: složitější
  - doporučení: pokud možno nepoužívat globální proměnné

Kopírování

Hodnotová sémantika: Inicializace/přiřazení je kopírování.

Implicitní kopírování

- všechny atributy jsou zkopírovány (inicializace/přiřazení)
- to je většinou to, co chceme
- co když chceme jiné chování? (proč chceme jiné chování?)

Kopírovací konstruktor

- popisuje, jak se objekt kopíruje **při inicializaci**
- syntax: `Object(const Object& object)`
  - konstruktor, bere *konstantní referenci* na objekt stejného typu
  - proč referenci?
  - proč konstantní?

Kopírování (pokr.)

Kopírovací přiřazovací operátor

- popisuje, jak se objekt kopíruje **při přiřazení**
- syntax `Object& operator=(const Object& object)`
  - přetížený operátor (o těch více později)
  - bere konstantní referenci na objekt stejného typu
  - vrací referenci na aktuální objekt (zvyk, doporučeno)

Zákaz kopírování – explicitně vymazaný kopírovací konstruktor a přiřazovací operátor

```
Object(const Object&) = delete;
Object& operator=(const Object&) = delete;
```

Kopírování (pokr.)

Kdy definovat explicitní kopírovací konstruktor/přiřazení?

- hluboká místo plytké kopie
  - použití: datové struktury (kontejnery apod.)
- správa zdrojů
  - paměť, soubory, zámky, vlákna, síťová spojení, grafické elementy
- registrace objektů
  - objekty se samy registrují/logují apod.
  - objekty s identitou
- zákaz kopírování
  - objekty, u nichž kopírování nedává smysl (např. některé zdroje)

[ukázka]

Více o správě zdrojů: příští přednáška

Kopírování (pokr.)

Vynechání kopií (copy elision)

- optimalizace překladače (povolená, ne zaručená)
- povoleno v určitých specifických případech
  - funkce bere parametr hodnotou a dostane dočasný objekt
  - funkce vrací lokální objekt hodnotou

[ukázky]

Pointa: Pokud v těle funkce hodlám dělat kopii parametru, pak je lépe brát jej rovnou hodnotou.

Destruktory

Konec života objektů

- lokální objekty: na konci bloku
- atributy objektů: zároveň s koncem života objektu, kterému patří
- globální objekty: na konci programu
- dynamicky alokované objekty: explicitně, zavoláním `delete` (příště)

Destruktor

- speciální metoda volaná na konci života objektu
- jméno destruktoru = vlnka ~ + jméno třídy
- vždy bez parametrů a bez návratové hodnoty

[ukázka]

Destruktory (pokr.)

Pořadí volání destruktorů

- opačné pořadí než volání konstruktorů
- destruktory atributů se volají po destruktoru hlavního objektu
- volání destruktorů při dědičnosti (později, v přednášce o OOP)

Všimněte si:

- volání destruktorů je **deterministické**
- víme přesně, kdy nastane konec života objektu
  - srovnejte s jinými jazyky
- tato vlastnost umožňuje princip RAII, o kterém bude řeč příště

[ukázka]

Pravidla

Rule of Zero

- pokud možno, nepište kopírovací konstruktor/přiřazení ani destruktor
- vhodné pro třídy, které nevlastní žádný zdroj (resource)

Rule of Three

- jakmile třída spravuje nějaký zdroj, pak je typicky třeba explicitně definovat všechny tři:
  - kopírovací konstruktor
  - kopírovací přiřazovací operátor
  - destruktor

Rule of Five (od C++11)

- přidává se ještě přesouvací (*move*) konstruktor/přiřazení
- vyžaduje pochopení tzv. *rvalue references*; nad rámec tohoto předmětu

Různé varianty: **Rule of three and a half**, **Rule of four and a half**

- tzv. *copy-and-swap* idiom (příští přednáška)

Závěrečný kvíz

https://kahoot.it