PB161 Programování v jazyce C++
Přednáška 4

Přetěžování funkcí
Konstruktory a destruktory
Lehký úvod do I/O

Nikola Beneš, Vladimír Štill

9. října 2018

Přetěžování funkcí

Přetěžování (overloading)

- různé funkce/metody se stejným jménem, ale různou definicí
- musí se lišit typem nebo počtem parametrů nebo `const` (u metod)
- Pozor! V C++ nestačí, pokud se liší pouze návratovým typem; proč?

```
void f(int x) {
    std::cout << "f s parametrem int: " << x << '\n';
}
void f() {
    std::cout << "f bez parametrů\n";
}
void f(double d) {
    std::cout << "f s parametrem double: " << d << '\n';
}
f(3);   // zavolá void f(int)
f(5.0); // zavolá void f(double)
f();    // zavolá void f()
```

lecture04_01.cpp, lecture04_02.cpp

Přetěžování (overloading) – pokr.

Pravidla pro přetěžování

- přesná shoda
- 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)

lecture04_03.cpp

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!
```

- poznámka: moderní kompilátory raději ohlásí chybu
- proto máme od C++11 speciální ukazatel **`nullptr`**

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

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

`lecture04_04.cpp`

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 */) {
    // ...
}
```

- implicitní parametry mohou být jen v první deklaraci
- pro čitelnost je zvykem je v komentářích zopakovat

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 (tzv. delegující konstruktory)

`lecture04_05.cpp`

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)

Kdy se volá konstruktor

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

`lecture04_06.cpp, lecture04_07.cpp`

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?

`lecture04_08.cpp`

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? Protože teprve definujeme, jak kopírovat.
  - proč konstantní?

`lecture04_08.cpp`

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? Protože teprve definujeme, jak kopírovat.
  - proč konstantní?
    Protože není slušné při kopírování měnit originál.

`lecture04_08.cpp`

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
    (ne nutně, viz *copy-and-swap* idiom, příští přednáška)
  - 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;
```

`lecture04_08.cpp`

Kopírování (pokr.)

Většinou nepotřebujeme definovat explicitní kopírovací konstruktor/přiřazení.

- defaultní kopírování často dělá to, co chceme

Kopírování (pokr.)

Většinou nepotřebujeme definovat explicitní kopírovací konstruktor/přiřazení.

- defaultní kopírování často dělá to, co chceme

Kdy definovat explicitní kopírovací konstruktor/přiřazení?

- hluboká místo plytké kopie
  - použití: vlastní 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)

Více o správě zdrojů: příští přednáška

`lecture04_09.cpp`

Kopírování (pokr.)

Vynechání kopií (copy elision)

- optimalizace překladače (povolená, od C++17 někdy i zaručená)
- povoleno v určitých specifických případech
  - funkce bere parametr hodnotou a dostane dočasný objekt
  - funkce vrací lokální objekt hodnotou

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

`lecture04_10.cpp, lecture04_11.cpp`

Destruktory

Konec života objektů

- lokální objekty: na konci bloku
- atributy objektů: zároveň[1] 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

[1] ve skutečnosti *těsně po*, viz další slajd

`lecture04_12.cpp`

Destruktory (pokr.)

Pořadí volání destruktorů

- opačné pořadí než volání konstruktorů[2]
- 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

`lecture04_13.cpp`

[2] to se týká sémantiky jazyka, ne chování standardní knihovny

Pravidla

Rule of Zero

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

Rule of Three

- jakmile třída spravuje nějaký zdroj, pak je typicky třeba explicitně definovat všechny tři (nebo alespoň některé zakázat):
  - kopírovací konstruktor
  - kopírovací přiřazovací operátor
  - destruktor

`lecture04_14.cpp`

Pravidla (pokr.)

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)

Lehký úvod do I/O

Už jsme viděli

- `std::cin`, `std::cout`, operátory `>>` a `<<`, `std::getline`

Souborový vstup/výstup `<fstream>`

```
std::ofstream output("output_file.txt");
std::ifstream input("input_file.txt");
```

- s proměnnými typu `std::ofstream` a `std::ifstream` smíme zacházet jako s `std::cout` a `std::cin`
- soubory se zavřou automaticky na konci bloku

`lecture04_15.cpp`

Lehký úvod do I/O (pokr.)

Řetězcový vstup/výstup `<sstream>`

- `std::istringstream` se inicializuje řetězcem, můžeme z něj pak číst stejně jako z `std::cin`
- do `std::ostringstream` můžeme zapisovat jako do `std::cout`, k výsledku přistoupíme pomocí metody `str()`

```
std::ostringstream out;
out << "Pi je " << 3.14 << '\n';
std::string result = out.str();
std::cout << result;

std::istringstream in("1 2 3 4 5");
int num;
while (in >> num) {
    std::cout << num << '\n';
}
```

`lecture04_16.cpp`

Lehký úvod do I/O (pokr.)

Chybové stavy I/O proudů

- metody `good()`, `fail()`, `bad()`, `eof()`
- proudy se umí přetypovat na `bool`
  - `true`/`false`, jestli je proud v dobrém stavu
- operátor `>>` i `std::getline` vrací vstupní proud

```
while (cin >> num) { /* ... */ }
while (std::getline(cin, str)) { /* ... */ }
```

Obecné proudy

- `std::istream`, `std::ostream` (bázové třídy – viz dědičnost, později)
- funkci s parametrem typu `std::istream&` můžeme předat libovolný vstupní proud (`std::cin`, proměnnou typu `std::ifstream` nebo `std::istringstream`)
- podobně pro `std::ostream&`

`lecture04_17.cpp`

Závěrečný kvíz

https://kahoot.it