

PB161 Programování v jazyce C++

Přednáška 6

Přátelství
Přetěžování operátorů
Vstup a výstup

Nikola Beneš

25. října 2016

Přátelství



Přístupová práva `private`, `protected`

- k čemu jsou dobrá?
- umožňují princip *zapouzdření* (encapsulation)
- zabraňují chybám (udržení konzistentního stavu objektu)

Někdy chceme tato práva porušit

- proč?
- iterátory
- operátory (uvidíme za chvíli)
- testování (možná)
- třídy, které spolu potřebují vzájemně interagovat na úrovni soukromých členů

Selektivní přístup: chci si vybrat, kdo může sahat na mé soukromé atributy

Klíčové slovo `friend`

- selektivní udělení přístupu k soukromým atributům a metodám
- přístup povoluje ten, k jehož soukromým členům má být přístupováno
- uvnitř deklarace třídy (kdekoliv)
- `friend class` `JmenoTridy`;
- `friend` typ `jmenoFunkce(parametry)`;
 - funkci je možno zároveň i definovat
(pak se považuje za `inline` funkci)

[ukázka]

Vlastnosti `friend`

- třída si určuje, kdo je její přítel, ne naopak
- přátelství není reciproční (symetrické)
- přátelství není tranzitivní
- přátelství není dědičné (o dědičnosti se budeme bavit příště)
 - vaši přátelé nejsou přáteli vašich dětí
 - přátelé vašich dětí nejsou vašimi přáteli

Přetěžování operátorů



Přetížené operátory (overloaded operators)

- už jste viděli, kde?
- téměř v každém jazyce (aritmetické operátory pro `int` vs. `float`)
- C++ umožňuje přetížit operátory pro vlastní datové typy
- už jsme viděli:
 - `operator=` pro kopírovací přiřazení
 - `operator+` pro řetězce

Proč přetěžovat operátory?

- zlepšit čitelnost kódu
- usnadnit používání třídy
- snížit chyby při použití třídy

Syntax

- volná funkce nebo metoda třídy
- **operator** a označení operátoru
- konkrétní syntaxe závisí na počtu parametrů
 - v případě metod je prvním operandem aktuální objekt

Zdroje

- https://en.wikipedia.org/wiki/Operators_in_C_and_C++
- <http://en.cppreference.com/w/cpp/language/operators>
- <http://stackoverflow.com/questions/4421706/operator-overloading>

Operátor jako funkce

- typicky **friend** (i když ne nutně)
- unární operátory – jeden parametr
- binární operátory – dva parametry

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

[ukázka použití]

Operátor jako metoda

- levý operand je `*this`
- počet parametrů: o jeden méně než implementace funkcí

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

[ukázka použití]

Přetížené operátory (pokr.)

Jak přetěžovat operátory?

- některé operátory není možno přetěžovat
 - `., ::, ? :`
- některé operátory musí být implementovány jako metody
 - `=, [], ->, ()`
- ostatní je možno implementovat jako metody nebo jako funkce

Doporučení

- pokud nemáme jak zasáhnout do třídy levého objektu: funkce
- unární operátor: metoda
- binární operátor, který se chová ke svým operandům stejně: funkce
- binární operátor, který se chová k levému operandu jinak: metoda
 - nezapomínejte na `const`

Omezení přetěžování operátorů

- nelze definovat nové operátory
- nelze měnit počet parametrů (kromě operátoru `()`)
- nelze definovat nové operátory pro primitivní typy
 - alespoň jeden z parametrů musí být uživatelsky definovaného typu
- nelze měnit prioritu ani asociativitu operátorů

Přiřazení `operator=`

- musí být metoda
- už jsme viděli v části o kopírování
- nezapomeňte na idiom *copy and swap*

```
X& X::operator=(X other) {  
    swap(other);  
    return *this;  
}
```

Přetěžování operátorů

Porovnávání `operator==`, `operator!=`, `operator<`, `operator<=`,
`operator>`, `operator>=`

- měly by vracet `bool`
- neměly by měnit své parametry (`const`)
- typicky jako funkce
- pokud přetížíte jeden, je vhodné přetížit i všechny ostatní

Přetěžování operátorů – porovnávání

```
bool operator<(const X& l, const X& r) {  
    // do the actual comparison  
    return ...;  
}  
  
bool operator>(const X& l, const X& r) { return r < l; }  
bool operator>=(const X& l, const X& r) { return !(l < r); }  
bool operator<=(const X& l, const X& r) { return !(r < l); }  
  
bool operator==(const X& l, const X& r) {  
    // do the actual comparison  
    return ...;  
}  
  
bool operator!=(const X& l, const X& r) { return !(l == r); }
```

[ukázka]

Přetěžování operátorů

Aritmetické operátory `operator+`, `operator+=` apod.

- silně doporučené: ke každému operátoru přetížit i kombinaci s přiřazením
- `+` apod. by měly být funkce a měly by vracet hodnotu
- `+=` apod. by měly být metody a měly by vracet referenci
- pokud chceme dělat kopii
 - může být dobrý nápad brát jeden z operandů (levý) hodnotou
 - typicky pak můžeme implementovat `+` pomocí `+=` apod.
- pokud chceme interakci i s jinými typy: pamatujte na symetrii

[ukázka]

Přetěžování operátorů

Indexování `operator[]`

- musí být metoda
- typicky chceme konstantní a nekonstantní přetížení

```
class X {  
    // ...  
public:  
    value_type& operator[] (index_type ix);  
    const value_type& operator[] (index_type ix) const;  
};
```

[ukázka]

Inkrement a dekrement `operator++`, `operator--`

- typicky jako metody
- dvě varianty: postfixová varianta má (nepoužívaný) parametr `int`
- prefixová varianta by měla vracet referenci
- postfixová varianta by měla vracet hodnotu (kopii)

[ukázka]

Přetěžování operátorů

Dereference `operator*`, `operator->`

- pro typy, které se chovají jako ukazatele (iterátory, chytré ukazatele)
- typicky jako metody
- `operator*` – vrací referenci
- `operator->` – vrací ukazatel nebo něco jako ukazatel (řetězení `->`)

$$a \rightarrow b \implies a.\text{operator->}() \rightarrow b$$

```
class MyIterator {  
    value_type* ptr;  
public:  
    value_type& operator*()          { return *ptr; }  
    const value_type& operator*() const { return *ptr; }  
    value_type* operator->()          { return ptr; }  
    const value_type* operator->() const { return ptr; }  
};
```

Přetěžování operátorů

Přetypování operator novytyp

- speciální operátor pro implicitní přetypování
- musí být metoda (bez parametrů)
- nepíše se návratová hodnota

```
class X {  
    int val;  
public:  
    X(int v) : val(v) {}  
    operator int() const { return val; }  
};  
  
int main() {  
    X x(3);  
    int a = x;  
    return x;  
}
```

Přetěžování operátorů – přetypování

Problémy s implicitním přetypováním

```
// která funkce se zavolá?
```

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

```
void f(X& x);
```

```
int main() {
```

```
    f(X(3));
```

```
}
```

- C++11: klíčové slovo **explicit**

```
class X {
```

```
// ...
```

```
public:
```

```
// ...
```

```
    explicit operator int() const { return val; }
```

```
};
```

Přetěžování operátorů – přetypování

Použití `explicit` pro `bool`

- překladač smí použít explicitní operátor `bool` pro přetypování, ale jen na `bool`, ne dál
- použití pro objekty, na jejichž stav se můžeme dotazovat pomocí `if`

```
class X {  
    // ...  
    bool is_ok;  
public:  
    // ...  
    explicit operator bool() const { return is_ok; }  
};  
  
int main() {  
    X x;  
    if (x) { /* ... */ }    // OK  
    int a = x;    // CHYBA  
}
```

Přetěžování operátorů

Funkční volání `operator()`

- musí být metoda
- může brát libovolný počet parametrů
- můžu tím vytvořit objekt, který se dá volat jako funkce
- použití: tzv. funkční objekty pro algoritmy

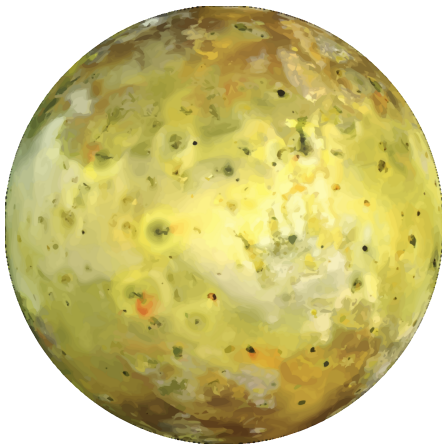
```
class Adder { // not the snake
    int val;
public:
    Adder(int v) : val(v) {}
    int operator()(int x) { return x + val; }
};
```

Přetěžování operátorů

Vstup a výstup `operator>>`, `operator<<`

- za chvíli

Vstupní/výstupní proudy



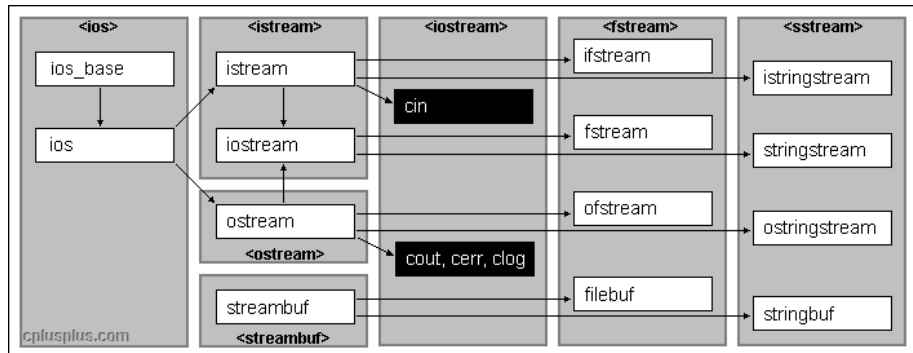
Vstup a výstup z různých druhů zařízení

- soubory
- klávesnice, obrazovka (terminál)
- tiskárna, skener
- síťový socket

Abstrakce konkrétního zařízení

- proudy (streams)
- data „plynou“ proudem od zdroje k cíli
- využívá principů OOP

Hierarchie I/O proudů



Standardní instance

- `cin` – standardní vstup, instance `istream` (`stdin` v C)
- `cout` – standardní výstup, instance `ostream` (`stdout` v C)
- `cerr` – standardní chybový výstup, instance `ostream` (`stderr` v C)
- `clog` – standardní logovací výstup, instance `ostream` (`stderr` v C)

- `cin` je svázaný s `cout`: jakýkoli vstup způsobí `cout.flush()`
- `cerr` je svázaný s `cout`: jakýkoli výstup způsobí `cout.flush()`
- `clog` není (defaultně) svázaný s `cout`

Operátor výstupu <<

```
int x = 7;
const double pi = 3.14;
std::string s = " is ";
cout << x << " + " << pi << s << x + pi << endl;
// 7 + 3.14 is 10.14
```

- pozor na prioritu operátorů
- přetížení << pro vlastní třídy

ostream& **operator**<<(ostream& out, **const** Object& object);

- přístup k **private** a **protected** atributům našeho objektu

[ukázka]

Vstup/výstup – operátory (pokr.)

Operátor vstupu >>

```
int x;  
double d;  
std::string s;  
  
std::cin >> x >> d;  
std::cin >> s; // přečte jedno slovo ze vstupu
```

- přetížení >> pro vlastní třídy

```
istream& operator>>(istream& in, Object& object);
```

[ukázka]

Chybové stavy proudů

- proudy obsahují příznaky naznačující chybu
 - eofbit, failbit, badbit
- metody pro testování stavu
 - good(), fail(), eof()
 - přetížené chování proudů jako **bool**, přetížený operátor !

```
int x;  
cin >> x;  
if (cin) { /* načtení hodnoty do x se povedlo */ }  
if (cin.good()) {  
    /* načtení hodnoty se povedlo a není konec souboru */  
}
```

- metoda pro vyčištění příznaků: clear()
- http://en.cppreference.com/w/cpp/io/ios_base/iostate

Proudy pro vstup/výstup ze souborů

- třídy ifstream, ofstream, fstream
- konstruktor se jménem souboru, metody open() a close()
- destruktork zavírá proud automaticky (RAII)

```
#include <fstream>
```

```
using namespace std;
```

```
int main() {  
    ofstream output("soubor.txt");  
    output << "Hello, world!\n";  
    output.close();  
    output.open("soubor2.txt");  
    output << 3.14 << endl;  
    // o uzavření se postará destruktork  
}
```


Mód otevření souboru (nepovinný parametr konstruktoru)

- binární příznaky (flags), kombinujeme pomocí |
- typ otevření
 - `ios::in` (vstup, implicitní pro `ifstream`)
 - `ios::out` (výstup, implicitní pro `ofstream`)
 - `ios::in | ios::out` (obojí, implicitní pro `fstream`)
- způsob otevření
 - `ios::binary` (binární data, implicitní jsou textová data)
 - `ios::app` (append, výstup na konec souboru)
 - `ios::trunc` (vymaže soubor)
- více viz <http://cppreference.com>
 - hledejte `ios` a `openmode`

[ukázka: příklad práce se soubory]

Třída `istream`

- operátor `>>` – už známe
- metoda `get()`
 - bez parametrů vrátí jeden znak
 - více znaků pomocí `get(buffer, length)` (čte do konce řádku)
- metoda `getline(buffer, length)`
 - podobně jako `get()`, ale zahodí znak konce řádku (`'\n'`)
- metoda `read(buffer, count)`
 - blokové čtení daného počtu bytů
 - hlavně pro binární soubory
- metoda `gcount()`
 - počet znaků načtených při posledním vstupu
- metoda `peek()`
 - náhled na další znak na vstup, bez jeho přečtení
- metoda `ignore(count, delim)`
 - zahodí *count* znaků ze vstupu (až po znak *delim*)
- a další ...
 - viz reference na webu

Třída `istream` (pokr.)

- načtení řádku do řetězce typu `std::string`
 - funkce `getline(istream &, string &)`

```
std::string s;  
getline(std::cin, s);
```

- součást knihovny `string`, ne knihovny `iostream`

- operátor << – už známe
- metoda `put()`
 - zapíše jeden znak
- metoda `write(output, count)`
 - protiklad `read()`
- zápis na konci souboru soubor zvětšuje
- zápis uvnitř souboru soubor přepisuje

Pozice a posun v souboru

- odkud se čte, kam se zapisuje?
 - „get“ ukazatel (třída `istream` a její potomci)
 - „put“ ukazatel (třída `ostream` a její potomci)
- `tellg()`, `tellp()` – získání pozice ukazatelů
- `seekg()`, `seekp()` – nastavení pozice ukazatelů; dva parametry
 - odkud:
 - `ios::beg` začátek souboru
 - `ios::cur` aktuální pozice
 - `ios::end` (konec souboru)
 - offset: o kolik se posunout od pozice *odkud*
- počáteční pozice v souboru – závisí na módu otevření

Vyrovnávací buffery

- data poslaná do proudu nemusí být ihned zapsána do cíle
- vyrovnávací paměť typu `streambuf` pro každý proud
- přenos z vyrovnávací paměti do cíle
 - při uzavření souboru (`close()`, destruktorka)
 - při zaplnění bufferu
 - explicitně pomocí manipulátorů (`endl`, `flush`, `sync`, `unitbuf`)

- speciální objekty, které je možno předávat operátorům << a >>
- flush – vyprázdní buffer
- endl – zapíše konec řádku a vyprázdní buffer
- dec, hex, oct – změni způsob reprezentace čísel
- setw, setfill, setprecision – mění formát vstupu a výstupu
- left, right – mění zarovnání

```
const double pi = 3.1415;  
cout << setw(10) << setfill('^') << left << setprecision(3)  
      << pi << endl;  
// výstup: 3.14^^^^
```

- <http://en.cppreference.com/w/cpp/io/manip>

Proudy dat v paměti

- třídy `stringstream`, `istringstream`, `ostringstream`
- konstruktor může brát řetězec – počáteční stav proudu
- metoda `str()` nastaví obsah proudu
 - bez parametrů vrátí aktuální obsah proudu jako `string`

[ukázka: příklad použití]

`std::istream_iterator<typ>`

- jeden parametr – vstupní proud
- procházení pomocí iterátoru je načítání dalších hodnot ze vstupu
- bez parametru: iterátor na konec (jakéhokoli) proudu

`std::ostream_iterator<typ>`

- parametry – výstupní proud, (nepovinný) oddělovač
- zapisování do iterátoru způsobí zápis na výstup

[ukázka použití]

<https://kahoot.it>

```
#include <fstream>
#include <string>

int main() {
    using namespace std;
    ifstream input("input.txt");
    ofstream output("output.txt");
    if (!input) { return 1; }
    if (!output) { return 2; }
    string line;
    getline(input, line);
    output << line << '\n';
    input.close();
}
```