PB161 Programování v jazyce C++
Přednáška 6

Přátelství
Přetěžování operátorů
Vstup a výstup

Nikola Beneš

23. října 2018

enum

Klasický `enum`

- funguje stejně jako v C
- žádný typová kontrola, implicitní přetypování
- navíc od C++11: bázový typ `enum X : short int { ... }`

`enum class` (od C++11)

- typová kontrola, přetypování pomocí `static_cast`

```
enum class Color { red, green, blue };

Color c = red; // chyba!
Color c = Color::red;
int x = c; // chyba!
int x = static_cast<int>(c);
```

- není možno napsat `using Color::red;`

Přátelství

Motivace

Přístupová práva `private`, `protected`

- k čemu jsou dobrá?

Motivace

Přístupová práva `private`, `protected`

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

Někdy chceme tato práva porušit

- proč?

Motivace

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 členy (atributy, metody)

Deklarace přátel

Klíčové slovo `friend`

- selektivní udělení přístupu k soukromým atributům a metodám
- přístup povoluje ten, k jehož soukromým členům má být přistupováno
- uvnitř deklarace třídy (kdekoliv)

- `friend class JmenoTridy;`
- `friend typ jmenoFunkce(parametry);`
  - funkci je možno zároveň i definovat
    (pak se považuje za `inline` funkci)

`lecture06_01.cpp`

Deklarace přátel (pokr.)

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 nutně přáteli vašich dětí
  - přátelé vašich dětí nejsou nutně vašimi přáteli

Přetěžování operátorů

Motivace

Přetížené operátory (overloaded operators)

- už jste viděli, kde?

Motivace

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

Přetížené operátory

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

Přetížené operátory (pokr.)

Operátor jako funkce

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

$$\texttt{a + b} \Longrightarrow \texttt{operator+(a, b)}$$

`lecture06_02.cpp`

Přetížené operátory (pokr.)

Operátor jako metoda

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

$$\texttt{a + b} \Longrightarrow \texttt{a.operator+(b)}$$

`lecture06_03.cpp`

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 objektu nalevo: 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`

Přetížené operátory (pokr.)

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

Přetěžování 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 ==, je vhodné přetížit i != a naopak
- pokud přetížíte < apod., 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); }
```

lecture06_04.cpp

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

`lecture06_05.cpp`

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;
};
```

lecture06_06.cpp

Přetěžování operátorů

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

- typicky jako metody
- dvě varianty: postfixová varianta má (nepoužívaný) parametr `int`
- prefixová varianta by měla vracet referenci na aktuální objekt
- postfixová varianta by měla vracet hodnotu (kopii objektu před modifikací)

`lecture06_07.cpp`

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 `->`)

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

```
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 novy_typ`

- 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;
}
```

lecture06_07.cpp

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 uvnitř podmínky `if`

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

`lecture06_08.cpp`

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; }
};
```

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

- za chvíli

`lecture06_08.cpp`

Vstupní/výstupní proudy

Motivace

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)
  - (defaultně) jediný nebufferovaný proud na tomto slajdu
- `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`

Vstup/výstup – operátory

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

Vstup/výstup – operátory

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

```
std::ostream& operator<<(std::ostream& out,
                         const Object& object);
```

- přístup k `private` a `protected` atributům našeho objektu
  - je-li třeba, použijte `friend`

`lecture06_09.cpp`

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
```

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

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

lecture06_10.cpp

Vstup/výstup – chybové stavy

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()`
- destruktor zavírá proud atomaticky (RAII)

```
#include <fstream>
using namespace std;

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

Souborové proudy (pokr.) <fstream>

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`
  - přehled: https://en.cppreference.com/w/cpp/io/basic_filebuf/open

`lecture06_11.cpp`

Třída `istream`

- operátor >> – už známe
- metoda `get()`
  - bez parametrů vrací 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 `std::getline(std::istream &, std::string &)`

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

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

Třída `ostream`

- 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()`, destruktor)
  - při zaplnění bufferu
  - explicitně pomocí manipulátorů (`endl`, `flush`, `sync`, `unitbuf`)

Manipulátory <iostream>, <iomanip>

- speciální objekty, které je možno předávat operátorům << a >>

- `flush` – vyprázdní buffer
- `endl` – zapíše konec řádku a vyprázdní buffer
- `dec`, `hex`, `oct` – změní 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 vrací aktuální obsah proudu jako `string`

`lecture06_12.cpp, lecture06_13.cpp`

Iterátory nad proudy

`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
- oddělovač vypíše i za posledním prvkem

`lecture06_14.cpp`

Závěrečný kvíz

https://kahoot.it

Závěrečný kvíz

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();
}
```