

PB161 – 5. přednáška (19. října 2015)

Jmenné prostory

Motivace

- problém: výskyt dvou entit (např. tříd) se stejným jménem
- typicky nenastává uvnitř jednoho projektu
- použití více nezávislých knihoven
 - příp. použití knihoven v existujícím projektu

```
// library1.h
```

```
class Object { /* ... */ };
```

```
// library2.h
```

```
class Object { /* ... */ };
```

```
// main.cpp
```

```
#include "library1.h"
```

```
#include "library2.h"
```

```
// při překladu main.cpp dojde k chybě  
// error: redefinition of 'class Object'
```

- jak se tomuto problému vyhnout?

Implicitní jmenné prostory

- oblasti platnosti identifikátorů
- vznikají automaticky
- uzavřeny mezi složené závorky
- deklarace tříd a struktur
- lokální jmenné prostory: funkce/metody, cykly, bloky (lokální rámce / local scope)
- globální jmenný prostor (globální rámec / global scope)
- v jednom jmenném prostoru nemohou být dvě entity se stejným jménem

```
// globální jmenný prostor
```

```
int x;
```

```
double x; // CHYBA!
```

```
class Example {
```

```
// jmenný prostor třídy Example
```

```

    float x;
public:
    void method() const {
        // jmenný prostor metody method()
        double x;
        for (int i = 0; i < 3; ++i) {
            // jmenný prostor for cyklu
            std::string x;
        }
        {
            // jmenný prostor vnořeného bloku
            char x;
        }
    }
};

```

- přístup ke jmenným prostorům pomocí operátoru ::
 - globální jméno ::x
 - jméno uvnitř třídy Example::x
 - k lokálním jmenným prostorům není přístup (nemají jméno)

```

#include <iostream>
using namespace std;

int x = 17;

class Example {
    int x;
public:
    Example() : x(29) {}
    void print() const {
        int x = 3;
        {
            int x = 9;
            cout << x << endl; // 9
            cout << Example::x << endl; // 29
            cout << ::x << endl; // 17
        }
        cout << x << endl; // 3
    }
};

int main() {
    Example e;
    e.print();
}

```

Explicitní jmenné prostory

- je možno vytvářet vlastní pojmenované jmenné prostory
- syntax: `namespace jmeno { ... }`
- jmenné prostory je možno i vnořovat

```
namespace MyLib {
    void print();
    namespace Experimental {
        void print();
    }
}

namespace MyLib {
    class Example {
    public:
        void print() const;
    };
}

int main() {
    MyLib::Example ex;
    ex.print();
    MyLib::print();
    MyLib::Experimental::print();
    print(); // CHYBA!
}
```

- už znáte jeden `namespace` používaný ve standardní knihovně

```
// typický obsah standardních hlavičkových souborů
namespace std {
// ...
}
```

- proto píšeme `std::cout`, `std::string` apod.

Zpřístupnění jmenného prostoru

- plná kvalifikace
 - `std::string`
- direktiva `using namespace jmeno_prostoru;`
 - vloží obsah *celého* jmenného prostoru do aktuálního jmenného prostoru
 - zvyšuje riziko konfliktů
 - analogie `import java.*` v Javě nebo `from library import *` v Pythonu

- nepoužívat v globálním rámci v hlavičkových souborech!

```
#include <string>

string s; // CHYBA!

void print() {
    using namespace std;
    string s; // OK
}

int main() {
    string s; // CHYBA!
}
```

- deklarace `using jmeno_prostoru::jmeno_entity`
 - vloží *pouze* odkazovanou entitu
 - má prioritu před `using namespace`

```
#include "libAdam.h"
#include "libEve.h"

using namespace Adam; // obsahuje funkci getApple();
using namespace Eve;   // taky obsahuje getApple();

getApple(); // CHYBA!

using Eve::getApple;

getApple(); // OK, volá se Eve::getApple();

#include <iostream>
using std::cout;
using std::endl;

int main() {
    cout << "Hello!" << endl;
}
```

- alias jmenného prostoru

```
namespace Widget = System::Window::Widget;

• using a using namespace v globálním prostoru
  – užívejte rozumně
```

- nikdy v hlavičkových souborech
- vždy až po všech `#include`
- `using` a `using namespace` můžeme také používat lokálně
 - ve funkcích/metodách
 - ve vnořených blocích
 - není možno používat přímo uvnitř třídy (class scope)

```
class X {
    using namespace std; // CHYBA!
public:
    void fun() {
        using namespace std; // OK
    }
};
```

Použití jmenných prostorů

- ve vlastních knihovnách

```
// cool_library.h
#ifndef COOL_LIBRARY_H
#define COOL_LIBRARY_H

namespace cool_library {

class Cool { /* ... */ };

}

#endif
```

- oddělení kolidujících jmen při použití cizích knihoven, které nepoužívají explicitní jmenné prostory

```
namespace Lib1 {
#include "library1.h" // obsahuje třídu System
}
namespace Lib2 {
#include "library2.h" // obsahuje třídu System
}

int main() {
    System s; // CHYBA!
    Lib1::System s1; // OK
    Lib2::System s2; // OK
}
```

Další čtení pro zvidané

- <http://en.cppreference.com/w/cpp/language/namespace>
- <http://en.cppreference.com/w/cpp/language/lookup>
 - qualified name lookup
 - unqualified name lookup
 - argument-dependent lookup

```
namespace Test {  
    int x;  
    void print(int y) {}  
}  
  
int main() {  
    print(x); // CHYBA!  
    Test::print(x); // CHYBA!  
    Test::print(Test::x); // OK  
    print(Test::x); // taky OK, argument-dependent lookup  
}
```

- umožňuje používání přetěžovaných operátorů v jiných jmenných prostorech a mnohé jiné věci

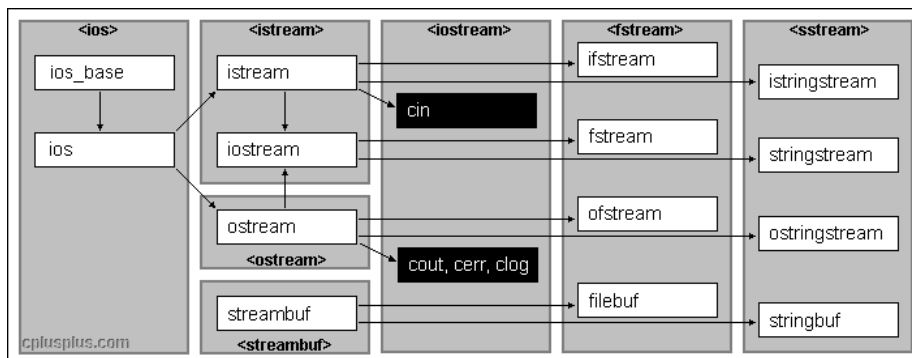
Vstupní/výstupní proudy

Motivace

- vstup a výstup z různých druhů zařízení
 - soubory
 - klávesnice, obrazovka (terminál)
 - tiskárna, skener
 - síťový socket
- abstrakce konkrétního zařízení
 - myšlenka proudů (streams)
 - data „plynou“ proudem od zdroje k cíli
 - využívá principů OOP

Hierarchie I/O proudů

- znázornění hierarchie C++ I/O knihovny (převzato z <http://cplusplus.com>)



- základní objekty v souboru `<iostream>`
- hierarchie umožňuje proudy s různými vlastnostmi
 - všimněte si vícenásobné dědičnosti
- abstraktní třída `ios_base`
- rozhraní pro vstupní proudy `istream`
- rozhraní pro výstupní proudy `ostream`
- proudy nelze kopírovat; proč?
 - v C++11 lze proudy *přesunovat* (move)

Standardní vstup/výstup

- hlavičkový soubor `<iostream>`
- standardní instance (objekty)
 - `cin` – standardní vstup, instance třídy `istream` (`stdin` v C)
 - `cout` – standardní výstup, instance třídy `ostream` (`stdout` v C)
 - `cerr` – standardní chybový výstup, instance třídy `ostream` (`stderr` v C)
 - `clog` – standardní logovací výstup, instance třídy `ostream` (`stderr` v C)
 - rozdíl mezi `cerr` a `clog`: `cerr` nepoužívá vyrovnávací paměť (buffer)
- speciální funkce pro manipulaci proudů (manipulátory)
 - např. `endl` – vloží konec řádku a zároveň provede vyprázdnění bufferů (flush)

Operátor výstupu `<<`

- zápis do výstupního proudu
 - přetížen pro standardní typy

```
int x = 7;
const double pi = 3.14;
char cString [] = " is ";
std::cout << x << " + " << pi << cString << x + pi << std::endl;
// 7 + 3.14 is 10.14
```

- pozor na prioritu operátorů
 - ve skutečnosti je << přetížený operátor bitového posuvu
 - http://en.cppreference.com/w/cpp/language/operator_precedence
- přetížení << pro vlastní třídy

```
ostream & operator<<(ostream & out, const Object & object) {
    // ...
}
```

- funkce, nikoli metoda (nemáme jak přidat metodu do třídy `ostream`)
- první parametr: reference na výstupní proud
 - nekonstantní: výstupem do proudu se mění jeho vnitřní stav
 - ne hodnotou: proudy se nesmí kopírovat
- druhý parametr: objekt, který chceme vypsát na výstup
 - typicky konstantní reference (nechceme objekt měnit ani kopírovat)
 - pro malé objekty možno použít i předání hodnotou
- operátor vrací referenci na výstupní proud
 - umožňuje řetězení výstupů

```
std::cout << 1 << 'x' << std::endl;
// se vlastně vyhodnotí jako
((std::cout << 1) << 'x') << std::endl;
// neboli
operator<<(operator<<(operator<<(std::cout, 1), 'x'), std::endl);
```

- takto definovaný operátor nemá přístup k `private` a `protected` atributům našeho objektu
 - řešení 0: gettery
 - řešení 1: speciální metoda `Object::print(std::ostream &)`
 - řešení 2: deklarace `friend` (později)

Příklad:

```
#include <iostream>

class Point {
    int x, y;
public:
    Point(int sx, int sy) : x(sx), y(sy) {}
    int getX() const { return x; }
    int getY() const { return y; }
};

std::ostream & operator<<(std::ostream & out, const Point & point) {
    out << '[' << point.getX() << ", " << point.getY() << ' ';
    return out;
}
```



```
} // dalo se napsat i na jeden řádek
```

```
int main() {  
    Point p(3,4);  
    std::cout << p << std::endl;  
    std::cerr << p << std::endl; // cerr je taky instance ostream  
}
```

Operátor vstupu >>

- čtení ze vstupního proudu
 - přetíženo pro standardní typy

```
int x;  
double d;  
char cString [50];
```

```
std::cin >> x >> d;  
std::cin >> cString; // možná problém
```

```
std::string s;
```

```
std::cin >> s; // přečte jedno slovo ze vstupu
```

```
// čtení celého řádku ze vstupu, viz metodu getline z <istream>  
// a funkci getline ze <string>
```

- přetížení >> pro vlastní třídy
 - podobně jako <<

```
istream & operator>>(istream & in, Object & object) {  
    // ...  
}
```

- druhý parametr: objekt, bereme referencí (budeme jej měnit)
- přístup k **private** a **protected** atributům našeho objektu
 - řešení 0: settery
 - řešení 1: konstruktor + kopírovací konstruktor
 - řešení 2: deklarace **friend** (později)

Příklad:

```
#include <iostream>
```

```
class Point {  
    int x, y;
```

```

public:
    Point(int sx, int sy) : x(sx), y(sy) {}
    void setX(int sx) { x = sx; }
    void setY(int sy) { y = sy; }
};

std::istream & operator>>(std::istream & in, Point & point) {
    int x, y;
    in >> x >> y;
    point.setX(x);
    point.setY(y);
    return in;
}

int main() {
    Point p(0,0);
    std::cin >> p;
    // v p teď jsou atributy nastaveny podle vstupu
}

```

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 zatím nedošlo ke konci vstupu (eof)
}

cin.clear(); // po vyřešení problému vyčistíme příznaky

• metoda pro vyčištění příznaků
    – clear()
• více viz http://en.cppreference.com/w/cpp/io/ios\_base/iostate
    – tabulka stavů dole

```

Souborové proudy

- hlavičkový soubor `<fstream>`
- třídy `ifstream`, `ofstream`, `fstream`
 - díky dědičnosti (viz tabulku nahoře) můžeme využívat známé chování z `cin` a `cout`
- konstruktor se jménem souboru, metody `open()` a `close()`

```
#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 zavření se postará destruktorka
}
```

- druhý (nepovinný) parametr konstruktoru – mód otevření souboru
 - 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> nebo <http://cplusplus.com/reference>
 - hledejte `ios` a `openmode`

Příklad: čtení souboru po znacích v C a C++

```
// C
#include <stdio.h>

int main(void) {
    FILE *file = NULL;
    char fileName [] = "soubor.txt";
    file = fopen(fileName, "r");
    if (file) {
        char c;
        while ((c = getc(file)) != EOF) {
            putchar(c);
        }
        fclose(file);
    }
```

```

    }
}

// C++
#include <fstream>
using namespace std;

int main() {
    ifstream file("soubor.txt");
    if (file.is_open()) {
        char c = cin.get();
        while (c.good()) {
            cout << c;
            c = cin.get();
        }
    }
}

```

Třída `istream` podrobně

- 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)`
 - zahodí `count` znaků ze vstupu
- a další ...
 - viz reference na webu
- načtení řádku do řetězce typu `std::string`
 - funkce `getline(istream &, string &)`

```

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

```

Třída `ostream` podrobně

- 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
 - to by nemuselo být efektivní (např. zápis do souboru)
- 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`)

Manipulátory

- speciální objekty, které je možno předávat operátorům `<<` a `>>`
- některé definovány v `<iostream>`, některé v `<iomanip>`
- `flush` – vyprázdní buffer
- `endl` – zapíše konec řádku a vyprázdní buffer
- `dec`, `hex`, `oct` – změni způsob reprezentace čísel (desítková, šestnáctková, osmičková)
- `setw`, `setfill`, `setprecision` – měni formát vstupu a výstupu
- `left`, `right` – měni zarovnání

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

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

Řetězcové proudy

- hlavičkový soubor `<sstream>`
- používají pro ukládání dat paměť místo souborů

- dědičnost: viz tabulku nahoře
 - umí všechno to, co standardní proudy
- třídy `stringstream`, `istringstream`, `ostringstream`
- konstruktor může brát řetězec – počáteční stav proudu
- metoda `str()` nastaví obsah proudu
 - bez parametrů vrací aktuální obsah proudu jako `string` Příklad použití:

```
#include <string>
#include <sstream>

using namespace std;

string toString(double d) { // tato metoda už je v C++11
    istringstream s;
    s << d;
    return s.str();
}

int main() {
    string s;
    getline(cin, s); // načtení řádku ze vstupu
    stringstream ss(s);
    int x, y;
    ss >> x >> y; // parsování řádku
}
```

Shrnutí

Jmenné prostory

- implicitní (lokální bloky, funkce, třídy, globální prostor)
- explicitní (`namespace`), mohou být vnořené
- zpřístupnění pomocí `using namespace` nebo `using`

Vstupní/výstupní proudy

- abstrakce skutečných zařízení
- hierarchie
- standardní vstup/výstup
- souborový vstup/výstup
- vstup/výstup v paměti
- přetížené operátory `<<` a `>>`
- další metody