

PB161 — 1. přednáška (21. září 2015)

Cíle předmětu

1. vysvětlit základy OOP
2. seznámit s možnostmi jazyka C++
3. (zavést a) podpořit praktické programátorské schopnosti
4. nadchnout do (nebo alespoň neodradit od) programování

Organizace předmětu

Přednášky

- nepovinné, ale snad přínosné a zábavné
- během jedné vnitrosemestrální test (datum bude upřesněno)
- zvané přednášky ke konci semestru (datum bude upřesněno)
- <http://cecko.eu/public/pb161>

Cvičení

- povinná, dvě neomluvené neúčasti tolerovány
- aktivní práce na příkladech, konzultace
- průběžné testíky formou odpovědníků
 - odpovědníky je možno řešit *pouze na cvičení*, jakékoli otevření odpovědníku mimo určenou dobu bude oceněno 5 zápornými body
- http://cecko.eu/public/pb161_cviceni
- možnost párového programování na cvičení – domluva s cvičícím

Ukončení předmětu

- zápočet: domácí úkoly + vnitrosemestrální test + testíky + úspěšné vypracování zápočtového příkladu na cvičení
- zkouška: zápočet + zkouškový test

Domácí úkoly

- 5+1 za semestr, zadávány průběžně (na webu cvičení)
- 12 bodů (9 za funkčnost, 3 za odevzdání) + bonusy
- deadline pro odevzdání na stránce úkolu (2 týdny)
- budou zveřejňována ukázková řešení
- možnost odevzdávání nanečisto (detaily na cvičení)
- odevzdávání do fakultního SVN, spuštění notifikačního skriptu
- max. 3 pokusy na odevzdání
- při kontrole používáme valgrind
 - pokud valgrind nalezne při testování chyby, bodové hodnocení se sníží až o dvě pětiny (40 %)
- *jednou z podmínek pro zápočet je mít alespoň 4 domácí úkoly s nenulovým bodovým hodnocením*

Shrnutí bodování

- tvrdé a měkké body
 - měkké body se nezapočítávají do limitu pro zápočet a úspěšné zvládnutí předmětu zkouškou; mohou pouze vylepšit výslednou známku

- domácí úkoly: 6×12
- bonusové části domácích úkolů: různé (měkké body)
- body za aktivitu na cvičeních: max 1b/cvičení (měkké body)
- body za odpovědníky na cvičeních: 10×3
- vnitrosestrální test: max 20b
- zápočtový příklad: není bodován, hodnocení splnil/nesplnil
- zkouškový test: max 80b

Hodnocení

- zápočet: alespoň 65 tvrdých bodů, splněna podmínka minimálního počtu 4 nenulových domácích úkolů, úspěšné splnění zápočtového příkladu
- zkouška: alespoň 95 tvrdých bodů a zápočet; známka podle součtu měkkých a tvrdých bodů:
 - alespoň 170 bodů: A
 - alespoň 150 bodů: B
 - alespoň 130 bodů: C
 - alespoň 110 bodů: D
 - alespoň 95 bodů: E

Materiály

- tyto podklady k přednáškám + slidy + ukázkové zdrojáky
- <http://cecko.eu/public/pb161>
- záznamy přednášek v ISu
- <http://cppreference.com>
- <http://cplusplus.com>

Kontakty

- přednášející: Nikola Beneš xbenes3@fi.muni.cz
 - konzultační hodiny: pondělí 14.10–15.40 B421
- cvičící: podle rozvrhu, konzultace na cvičeních
- studentští konzultanti
 - dodatečné konzultace nezávislé na skupině
 - konzultační hodiny na webu cecko.eu

Varování

- opisování domácích úkolů a *zveřejňování svého řešení* bude trestáno 0 body za daný úkol
 - nezapomeňte na podmínku 4 nenulových domácích úkolů
 - *je zakázáno zveřejňovat své řešení i po deadline domácího úkolu*
- provádíme automatickou kontrolu opisování u všech odevzdaných příkladů
 - každý s každým
 - každý s řešeními z minulých let (u podobných domácích úkolů)
 - podezřelé případy řeší manuální kontrola

Zpětná vazba

- předmětová anketa v ISu
- občasné dotazníky (obtížnost úloh, apod.)
- osobně, e-mailem
- krabice pro anonymní vzkazy (od 3. týdne semestru)

Programovací jazyk C++

Historie a vývoj

- 1979 Bjarne Stroustrup: C with Classes
 - původně rozšíření C o objektově-orientované prvky
 - vývoj C šel dál (C89, C99, C11), C a C++ se vzájemně ovlivňují
- 1983 C++
- 1998 ISO standard C++98
- 2003 ISO standard C++03 (drobné opravy)
- 2011 ISO standard C++11 (velké změny)
- máme rok 2015, kompilátory už podporují většinu vlastností C++11, proto i my budeme používat C++11
 - zdaleka ne všechny novinky, ale ty, které nám usnadní práci ano
 - u konstruktů, které nejsou podporovány C++03 bude upozornění a vysvětlení
 - speciální zvaná přednáška o C++11 ke konci semestru
- další vývoj: C++14, C++17?
- nestandardizovaná rozšíření:
 - užitečná rozšíření dosud nezahrnutá v normě
 - omezení přenositelnosti kódu, svázanost s konkrétní platformou
 - my budeme psát v souladu s normou (C++11)

Charakteristika C++

- imperativní, staticky typovaný jazyk
- objektově-orientovaný, s funkcionálními prvky (zejména od C++11)
- umožňuje generické programování (šablony)
- částečně zpětně kompatibilní s C, kód v C je často validní i v C++
 - https://en.wikipedia.org/wiki/Compatibility_of_C_and_C++
 - pro některé konstrukce z C má C++ lepší prostředky; např. nebude me používat malloc/free, ale new/delete
- rozsáhlá standardní knihovna
 - umožňuje psát kratší a přehlednější kód

Proč používat C++?

- široké rozšíření
- vysoká rychlost kódu
- vhodné pro:
 - větší projekty
 - systémové aplikace
 - rychlou grafiku
 - embedded zařízení
- spíše nevhodné pro:
 - webové aplikace
 - rychlé prototypy
- <http://benchmarksgame.alioth.debian.org>

Objektově-orientované programování

Základní představa OOP

- svět se skládá z objektů; všechno je objekt
- objekty mají svůj interní stav, který není vidět
- objekty komunikují pomocí zpráv
- objekty se často vytvářejí podle předem daného vzoru (třída, prototyp, ...)

Příklad: Představme si, že navrhujeme strategickou hru, v níž proti sobě bojují dvě armády. Jako objekty můžeme chápat jednotlivé jednotky. Hráč posílá jednotkám zprávy (pohni se, zaútoč, ...) a jednotky si posílají zprávy mezi sebou (zasáhl jsem tě).

Principy OOP

- zapouzdření
- abstrakce
- dědičnost
- polymorfismus

Zapouzdření

- objekt v sobě obsahuje jak data, tak i kód, který s těmito daty pracuje
- to umožňuje skrýt vnitřní reprezentaci dat
- objekt navenek komunikuje jen skrz své rozhraní
- výhody:
 - vnitřní implementace objektu se může změnit bez nutnosti změny ostatních objektů
 - ochrana proti chybám, uživatel objektu nemůže omylem změnit nějakou jeho kritickou část
 - zapouzdření jako způsob návrhu: nutí programátora rozvrhnout program do nezávislých částí
 - umožňuje další OOP vlastnosti

Abstrakce

- úzce souvisí se zapouzdřením, jde ale o obecnější princip
- umožňuje programátorovi pracovat s ideální představou, která není zatížena konkrétními implementačními detaily
- datová abstrakce
 - použití dat bez znalosti jejich reprezentace či skutečného umístění
 - příklad: databáze jako soubor na disku nebo na vzdáleném serveru
- funkční abstrakce
 - funkce může být použita, aniž by uživatel znal detaily její implementace
 - příklad: reakce na zprávu „zaútoč“ bude jiná u lučistníka a u rytíře, ale přesto jde o stejnou zprávu

Dědičnost

- objekt (třída) může dědit od jiného objektu (třídy)
- potomek může využít kódu rodiče
- takto můžeme vytvářet hierarchii objektů (tříd)
- snižuje nutnost opakování kódu
 - příklad: Lučistník i šermíř jsou pozemní jednotky, proto dává smysl mít třídu reprezentující pozemní jednotky, která obsahuje kód pro pohyb, od níž pak lučistník a šermíř dědí. Tak není třeba psát kód pro pohyb vícekrát.

- Liskovové princip nahraditelnosti (substitution principle):
 - potomek by měl vždy moci zastoupit předka
 - příklad: pokud něco umí pozemní jednotka (třeba se nalodit), musí to umět i lučištník
- alternativou k dědění (inheritance) je skládání (composition) – později

Polymorfismus

- souvisí s abstrakcí
- umožňuje psát kód obecně, pouze se znalostí určitého rozhraní
- polymorfismus skrze dědičnost
 - příklad: Máme seznam pozemních jednotek (třeba těch, co jsou aktuálně označené) a můžeme jim všem poslat příkaz k pohybu na určité místo, nezávisle na tom, že jednotlivé jednotky jsou různé.
- existuje více způsobů realizace polymorfismu (generické programování, ...)
 - o generickém programování si více řekneme v přednášce o šablonách

OOP jako styl myšlení; Object-Oriented Design

- OOP je především způsob přístupu k řešení
 - objektový návrh: co jsou objekty, jaké jsou mezi nimi vztahy
- můžeme programovat objektově i v neobjektových jazycích
- ale v objektově orientovaných jazycích k tomu máme syntaktickou podporu

Implementace příkladu v C: Vraťme se k představě strategické hry. Každou jednotku bychom v čistém C mohli reprezentovat jako `struct`, posílání zpráv bychom pak mohli implementovat pomocí funkcí:

```
struct Archer {
    int hit_points;
    int range;
    int x, y;    // position
};

void move(struct Archer * archer, int targetX, int targetY) {
    // ... code to move archer ...
}
```

Implementace OOP v C++

Třídy, metody, objekty

- přístup k OOP v C++ je založen na třídách
- třída (`class`) je datový typ, rozšíření datového typu `struct` z C
 - ve skutečnosti máme v C++ i `struct`
 - liší se jen v implicitních přístupových právech
- kromě atributů (dat) může obsahovat i metody (funkce)
- přístupová práva
 - `public` části vidí všichni
 - `protected` vidí jen potomci třídy (viz dědičnost)
 - `private` vidí jen třída sama
 - implicitní práva (pokud nic neuvedeme): u `class` je to `private`, u `struct` je to `public`

- metody realizují princip posílání zpráv
- objekty jsou konkrétní instance tříd

```
class LandUnit {
private: // soukromé, není vidět z venku
    int x, y;
public: // veřejné, je součástí veřejného rozhraní třídy
    void move(int targetX, int targetY); // jen prototyp
};

void LandUnit::move(int targetX, int targetY) {
    // ... kód pro pohyb jednotky ...
    x = targetX;
    y = targetY;
}

int main() {
    LandUnit soldier1; // soldier1 je objekt, instance LandUnit

    soldier1.move(1, 1);
}
```

- ve skutečnosti mají metody jeden skrytý parametr `this`
 - ukazatel na konkrétní objekt
 - typ metody `move` je jakoby

```
void LandUnit::move(LandUnit * this, int, int);
```

(pozor, toto není skutečné C++, `this` je implicitní parametr)
 - volání `soldier1.move(1, 1);` je pak jakoby

```
LandUnit::move(&soldier1, 1, 1);
```

(pozor, toto není skutečné C++, jen ilustrace toho, jak funguje volání metody)
 - použití atributů uvnitř metody: místo `x` a `y` jsme mohli psát

```
this->x
```

 a `this->y`
 - hodí se pokud má nějaký parametr metody shodné jméno s atributem třídy
- deklarace (prototypy) metod a jejich definice (implementace) je možno buď psát odděleně, jak je uvedeno výše, nebo i uvnitř deklarace třídy
 - vhodné pro menší metody (např. metody typu *getter* a *setter*)
 - překladač se k nim chová, jako by byly `inline`

```
class Person {
private:
    int age;
public:
    void setAge(int age) {
        this->age = age; // všimněte si použití ukazatele this
    }
    int getAge() {
        return age;
    }
}
```

```
    }
};
```

- typické rozvržení do souborů: deklarace třídy (a inline metody) do hlavičkového souboru (*.h), definice (implementace) metod do zdrojového souboru *.cpp
- zamyšlení: Co všechno patří k třídě? <http://got.ca/publications/mill02.html> (k třídě patří i volné funkce s danou třídou pracující)

Konstruktor

- motivace: třída má několik atributů, jaká bude jejich hodnota při vytvoření objektu?
 - neinicializovaná, což je nepříjemné
- řešení (neelegantní): metoda pro inicializaci atributů, kterou je třeba vždy zavolat
- lepší řešení: konstruktor
 - konstruktor je automaticky volán při vytváření objektu
 - zajišťuje, aby byl objekt od začátku v konzistentním stavu
 - kromě inicializace atributů může také vykonávat jiné činnosti (otevření souboru, spojení se serverem, ...)
 - může mít argumenty a může být přetížen (více konstruktorů s různými argumenty)
- syntaxe konstruktoru
 - stejné jméno jako třída
 - nemá návratovou hodnotu (ani `void`!)

```
class LandUnit {
    int x, y;    // implicitní práva jsou private
public:
    LandUnit();
    LandUnit(int, int); // jména atributů při deklaraci
                        // nejsou nutná
};

LandUnit::LandUnit() {
    x = 0;
    y = 0;
}

LandUnit::LandUnit(int x, int y) {
    this->x = x;
    this->y = y;
}

int main() {
    LandUnit soldier1; // volá se konstruktor bez parametrů
    LandUnit soldier2(17, 42); // volá se konstruktor s parametry
}
```

- pozor: Není možno psát `LandUnit soldier1();`, to totiž překladač chápe

jako deklaraci funkce s návratovým typem `LandUnit`; příště si povíme o nové syntaxi inicializace v C++11, která tento problém řeší

- stejně jako metody, i konstruktor může být definován uvnitř deklarace třídy
- inicializační sekce konstrukturu
 - inicializace atributů
 - předání parametrů konstrukturu rodiče (viz dědičnost)
 - lepší způsob inicializace než přiřazení: nemusí se vytvářet lokální kopie, nelze jinak před argumenty konstrukturu předka, nelze jinak inicializovat argumenty s referenčním typem (o referencích si řekneme později)

```
class LandUnit {  
    // ...jako předtím...  
    LandUnit(int tx, int ty) : x(tx), y(ty) {}  
    // všimněte si prázdného těla konstrukturu  
};
```

- co když nedefinuju žádný konstruktor?
 - vytvoří se tzv. defaultní konstruktor, který nechá atributy neinicializované
 - přesněji: defaultní konstruktor volá defaultní nebo bezparametrické konstruktory atributů; pro primitivní typy to znamená, že zůstanou neinicializované

Destruktory

- určeny pro úklid objektu poté, co přestal existovat
 - uvolnění dynamické paměti
 - zavření souborů, ukončení spojení apod.
 - automaticky volán při uvolňování objektu
 - konec platnosti lokální proměnné při statické alokaci
 - po zavolání `delete` při dynamické alokaci (později)
 - nemá žádné parametry, nelze jej přetěžovat, nevrací žádnou hodnotu
 - syntaxe: `~jméno třídy()`
 - destruktory se dá volat i explicitně, ale je to používáno jen ve velmi specifických situacích, v tomto předmětu se s tím nesetkáme
- ```
C++ class LandUnit { // ... virtual ~LandUnit(); // virtual bude vysvětleno u dědičnosti };
```

#### Dědičnost

- třída může dědit od jiné třídy
- v této přednášce jen úplné základy, více si řekneme příště:
  - zde používáme **public** dědičnost (všechna práva zděděných atributů a metod zůstanou stejná)

```
class Archer : public LandUnit {
 int range;
public:
 Archer(int x, int y, int r) : LandUnit(x, y), range(r) {}
 // při volání konstrukturu Archer se zavolá konstruktor
 // předka LandUnit
```



```

 void shootAt(int x, int y);
}

```

- nyní můžeme psát třeba:

```

Archer john(17, 42, 6);
john.move(20, 15); // používáme zděděnou metodu
john.shootAt(20, 10);

```

- ukazateli na předka můžeme předat potomka:

```

LandUnit * unit = &john;
unit->move(10, 7);
unit->shootAt(1, 1); // CHYBA! třída LandUnit nemá
 // metodu shootAt

```

- polymorfismus: dejme tomu, že i třídy Swordsman a Knight jsou potomky LandUnit; potom můžeme psát:

```

Swordsman joe(5, 5);
Knight lancelet(7, 7);
LandUnit * units[3] = { &john, &joe, &lancelet };
for (int i = 0; i < 3; ++i) {
 units[i]->move(i, 15);
}

```

- klíčové slovo **virtual**
  - určuje tzv. pozdní vazbu
  - pozdní vazba umožňuje polymorfismus, ale je dražší (pomalejší)
- destruktor třídy, z níž hodláme dědit, by měl být buď veřejný a virtuální, nebo **protected** a nevirtuální; více si o tom řekneme příště

```

class X {
public:
 void metodaA();
 virtual void metodaB();
};

```

```

class Y : public X {
public:
 void metodaA();
 void metodaB();
};

```

```

int main() {
 Y y;
 X * px = &y;
 px->metodaA(); // zavolá se X::metodaA(), časná vazba
 px->metodaB(); // zavolá se Y::metodaB(), pozdní vazba
}

```

- abstraktní třída je třída, která neimplementuje nějakou (virtuální) metodu
  - není možno vytvářet objekty abstraktní třídy
  - funguje tedy jen jako součást objektové hierarchie

```
class LandUnit {
 // ...
 virtual void attack(int x, int y) = 0;
 // speciální syntax "= 0" znamená, že metoda je čistě virtuální
};

void Archer::attack(int x, int y) {
 // kód pro útok
}

int main() {
 LandUnit soldier; // chyba!
 LandUnit * unit; // ukazatel na abstraktní třídu je v pořádku
 Archer john(9, 15);
 unit = &archer;
 unit->attack(10, 7);
}
```

## Shrnutí

### Objektově-orientované programování

- způsob návrhu řešení problémů pomocí rozdělení na víceméně autonomní celky – objekty
- principy OOP:
  - zapouzdření
  - abstrakce
  - dědičnost
  - polymorfismus

### OOP v C++

- třídy – obsahují atributy (data), metody, konstruktory, destruktory,
- přístupová práva – public, protected, private
- metody mají skrytý parametr **this**
- konstruktor slouží k inicializaci objektu
- destruktory slouží k úklidu

## Samostudium na příští týden

- najděte si na <http://cppreference.com> dokumentaci k typu **string** a naučte se jej používat
- najděte si na <http://cplusplus.com> tutoriál Basic Input/Output a naučte se základy vstupu a výstupu v C++
- dotazy na použití **stringu** mohou být součástí testíku na druhém cvičení, stejně tak i vstup a výstup pomocí **cin** a **cout**