

DĚDIČNOST, POLYMORFISMUS, POZDNÍ VAZBA

PB161 PROGRAMOVÁNÍ V JAZYCE C++

Nikola Beneš

21. března 2023

Actually, I made up the term object-oriented, and I can tell you I did not have C++ in mind. (Alan Kay)

<https://softwareengineering.stackexchange.com/q/46592>

Viditelnost položek vlastního typu

- **public** vidí všichni (pro **struct** implicitní)
- **private** vidí jen metody daného typu a *přátelé*
- **protected** vidí metody daného typu, metody jeho podtypů (potomků, viz dále; i nepřímých), a *přátelé*

Syntax

```
struct example {  
public: // not needed  
    // everything here is public  
private:  
    // everything here is private  
};
```



Přátelé

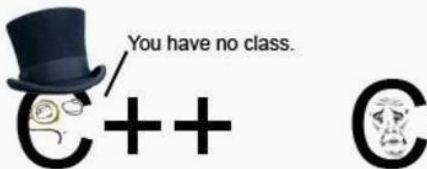
- přátelské funkce – už jsme viděli
- přátelské typy – **friend struct** typ;
nebo jen **friend** typ; (pokud byl typ deklarován dříve)
- přátelské metody – **friend void** typ::method(**int**);

Přátelství **není**

- reciproční (symetrické)
- tranzitivní
- dědičné
 - vaši přátelé nejsou nutně přáteli vašich dětí
 - přátelé vašich dětí nejsou nutně vašimi přáteli

Friends, much as in real life, are often more trouble than their worth.

(Scott Meyers)



Klíčové slovo **class**

- v C++ prakticky totéž, co **struct**
- rozdíl v implicitních přístupových právech a dědičnosti
 - **struct** – **public**
 - **class** – **private**
- to je vše

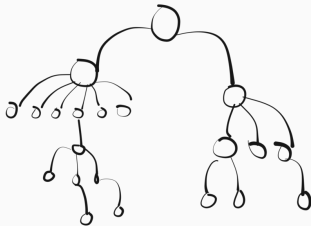
Princip polymorfismu

- obecný kód, který funguje s různými typy

Realizace polymorfismu

- *ad-hoc* polymorfismus
 - přetěžování funkcí
- *parametrický polymorfismus*
 - generické programování – v C++ **auto** parametry, šablony
 - častý ve funkcionálních jazycích
- *strukturální polymorfismus*
 - statický – deklarace, jaké funkce jsou potřeba
 - dynamický – ověření za běhu (*duck typing*)
 - „if it walks like a duck and quacks like a duck, it is a duck“
- *podtypový polymorfismus*
 - dědičnost – **dnešní téma**

DĚDIČNOST (PODTYPOVÝ POLYMORFISMUS)



```
class teacher : public employee {  
    // ...  
};
```

- *veřejná dědičnost* – vztah IS-A
 - jiné druhy dědičnosti mimo záběr předmětu
 - u **struct** implicitní (netřeba psát **public**)
- třída **teacher** *dědí* od třídy **employee**
 - **employee** je předkem (nadtypem) **teacher**
 - **teacher** je potomkem (podtypem) **employee**
- **teacher** má všechny položky / metody třídy **employee**
 - *objekty* typu **teacher** obsahují *podobjekt* typu **employee** (tzv. *bázový podobjekt*, *base subobject*)

Dědičnost (IS-A)

- C++: veřejná (**public**) dědičnost
- vztah podtyp–nadtyp
- *Liskovové substituční princip*: potomek může zastoupit předka
- umožňuje *polymorfismus*: obecný kód pro různé skutečné typy

Kompozice (HAS-A)

- objekty obsahují jiné objekty (položky třídy)
- často preferovaná před dědičností
 - otázka správného návrhu: IS-A nebo HAS-A?

```
class laptop : public cpu, public ram { ... };  
class stack : public vector { ... };  
class properties : public hash_table { ... };  
class square : public rectangle { ... }; // ???  
class computer_with_printer : public computer { ... };
```



Dědičnost a přístupová práva

- metody potomka mohou přistupovat pouze k **public** a **protected** položkám předka
- zvenku je možno přistupovat ke všem **public** položkám
 - nejsou-li skryté

Skrývání položek (*hiding*)

- existuje-li v potomkovi položka stejného jména jako v předkovi, položka předka je skrytá
 - týká se i přetížených metod
- explicitní přístup pomocí operátoru `::`

Konstruktory

- pořadí inicializace
 - inicializace předka
 - inicializace položek v pořadí deklarace
 - provedení těla konstrukturu
- implicitní bezparametrický konstruktor používá konstruktor předka bez parametrů
 - není-li žádný takový, implicitní konstruktor se nevytvoří
- ovlivnění inicializace předka – v inicializační sekci
 - `derived(...) : base{ ... }, ... { ... }`

Dědění konstruktorů

- konstrukce **using** `base::base;`
- vytvoří konstruktory potomka se stejnými hlavičkami jako konstruktory předka
- předají všechny parametry odpovídajícímu konstrukturu předka
- ostatní položky inicializují jako implicitní konstruktor

Kopírování / přesun

- implicitní speciální metody (konstruktory / operátor =) volají tutéž speciální metodu předka
 - (jakoby předek byl první položka typu)
- explicitní – pamatujte na *Rule of Zero / Rule of Five*
- (problémy s kopírováním polymorfních objektů – za chvíli)

Destruktory

- pořadí destrukce – opačné k pořadí inicializace
 - provedení těla destrukturu
 - destrukce položek v opačném pořadí deklarace
 - destrukce předka

Statický typ

- to, co je vidět v době kompilace
- typ deklarace proměnné, návratové hodnoty funkce apod.

Dynamický typ

- „skutečný“ typ objektu
- pro *hodnotové proměnné* je vždy totožný se statickým typem
- může se lišit u *referencí*
 - (včetně těch, co vzniknou dereferencí ukazatele)

```
dog fido;  
animal& ref = fido;  
animal* ptr = &fido;
```

Časná vazba (*early binding*)

- v C++ implicitní
- volá se metoda podle *statického typu* výrazu
 - tj. známá v době překladač
- jednodušší, v kódu přímo adresa metody

Pozdní vazba (*late binding*)

- volá se metoda podle *dynamického typu* objektu
- v C++ klíčové slovo **virtual**
- každá třída má tabulku virtuálních metod
- každý objekt si s sebou nese ukazatel do správné tabulky
 - týká se jen těch, co mají nějakou virtuální metodu
- pomalejší, adresu správné metody je třeba získat za běhu

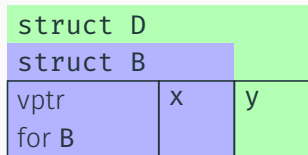
<https://gcc.godbolt.org/z/dnY9Y8os6>

Virtuální metody

- v předkovi označíme metodu slovem **virtual**
- metoda se *stejným typem včetně specifkátorů* v potomkovi neskrývá, ale nahrazuje (*override*)
 - je-li návratový typ ukazatel nebo reference, může nahrazující metoda vracet ukazatel nebo referenci na podtyp, tj. místo `base&` může vracet `derived&`
- v potomkovi není třeba psát **virtual**
 - metoda nemůže „přestat být virtuální“
 - od C++11 doporučeno psát **override** – překladač hlídá typ

TABULKA VIRTUÁLNÍCH METOD (ZJEDNODUŠENĚ)

```
struct B { virtual ~B() {}; virtual void f();  
          virtual void g(); int x; };  
struct D : B { void f() override; int y; };
```



umístěno ve statické paměti
vtable for D

type info for D
D::~~D()
D::f()
B::g()

vtable for B

type info for B
B::~~B()
B::f()
B::g()

Pozor!

- dealokace skrze ukazatel na předka je **nedefinované chování**, pokud předek nemá *virtuální destruktor*
 - týká se explicitního **new / delete**
 - týká se **std::unique_ptr** (s implicitním dealokátorem)
 - netýká se **std::shared_ptr**¹ (drží si extra informaci)

Doporučení

- máme-li polymorfní hierarchii (s virtuálními metodami), měli bychom mít virtuální destruktor
 - typicky není třeba měnit jeho chování, stačí = **default**
- alternativně: **protected** nevirtuální destruktor
 - pokus o dealokaci skrze předka je chyba při kompilaci

¹je-li dynamický typ znám při inicializaci: vytvoření pomocí **std::make_shared** nebo inicializací ukazatelem se správným typem (dynamický = statický)

Ukazatel `this`

- statický typ je vždy ukazatel na třídu, v níž se metoda nachází
 - případně `const` podle specifikátoru
- dynamický typ je skutečný typ objektu – *během jeho života*
- dynamický typ *během inicializace / destrukce* je vždy shodný se statickým typem
 - během inicializace předka ještě objekt potomka neexistuje (jeho položky ještě nebyly inicializovány, tělo jeho konstruktor nebylo provedeno)
 - během destrukce předka už objekt potomka neexistuje (jeho položky už byly destruovány, tělo jeho destruktoru už bylo provedeno)
 - *důsledek*: virtuální metody volané během inicializace / destrukce se nechovají jako virtuální

```
cat felix;  
animal pet = felix;
```

- co se stane?
- nemůže se zkopírovat celý objekt typu `cat`
 - dynamický typ proměnné `pet` je `animal`
 - velikost objektů je pevná, celý objekt typu `cat` se do proměnné `pet` ani *nemusí vejít*
- zkopíruje se část objektu potomka odpovídající předkovi
 - konkrétněji: bazový podobjekt
 - tzv. *slicing* („ořezávání“ objektů)

Důsledek

- Liskovové substituční princip funguje v C++ s referencemi a ukazateli, ale ne s předáváním objektů *hodnotou*

Položky polymorfních typů

```
struct bad_wizard {  
    animal familiar;  
};
```

```
struct good_wizard {  
    std::unique_ptr<animal> familiar;  
};
```

- chceme-li *vlastnit* položku polymorfního typu, musí být alokována dynamicky
 - ideálně s pomocí `std::unique_ptr`

Prvky kontejnerů

```
std::vector<animal> animals;  
animals.push_back(cat());  
animals.push_back(dog());
```

- co se stane?
- prvky vektoru jsou *hodnoty*, dochází k ořezávání (*slicing*)
- kontejnery polymorfních objektů musí používat ukazatele
- má-li kontejner objekty zároveň vlastnit, *musí být alokovány dynamicky*
 - ideálně s pomocí `std::unique_ptr`

Kopírování

- kopírovací konstruktory i přiřazovací operátory jsou *nevirtuální*
- idiom „virtuálního kopírovacího konstrukturu“

```
struct animal {  
    virtual std::unique_ptr<animal> clone() const {  
        return std::make_unique<animal>(*this);  
    }  
};  
  
struct dog : animal {  
    std::unique_ptr<animal> clone() const override {  
        return std::make_unique<dog>(*this);  
    }  
};
```


Polymorfní objekty

- přistupujeme k nim skrze ukazatele / reference na předka
- nemůžeme s nimi zacházet jako s „normálními“ objekty
 - položky tříd
 - prvky kontejnerů
 - kopírování
- alokujeme je dynamicky a „schováváme“ je za (chytré) ukazatele

Všimněte si

- některé jazyky se těmto problémům vyhýbají tím, že za ukazatele „schovávají“ **všechny** objekty

Čistě virtuální metoda

- metoda bez implementace v bázevé třídě
- notace `typ metoda(parametry) = 0;`

Abstraktní třída

- má alespoň jednu čistě virtuální metodu
- nelze od ní vytvářet instance
- *čistě abstraktní* – všechny metody čistě virtuální
 - s výjimkou destrukturu

Rozhraní (*interface*)

- čistě abstraktní třída bez položek
- nedrží data, jen deklaruje metody
- v C++ není klíčové slovo, jen dohoda

- v C++ je možno dědit od více tříd
 - `class C : public A, public B { ... };`
- bez problémů, pokud dědíme pouze od rozhraní;
případně rozhraní + jedna jiná třída

Problémy

- položky, metody stejného jména v třídách předků
- vzniká zejména tehdy, dědí-li třídy předků od stejné báze třídy
 - tzv. *diamond problém*
 - řeší se tzv. virtuální dědičností (mimo záběr předmětu)

```
class teacher : public person { ... };
```

```
class student : public person { ... };
```

- co když je někdo zároveň studentem i učitelem?
- alternativní řešení: kompozice místo dědičnosti – role

Typové konverze v objektové hierarchii

- implicitní: ukazatel na potomka → ukazatel na předka
 - totéž pro reference
- **static_cast**: ukazatel na předka → ukazatel na potomka
 - totéž pro reference
 - **potenciálně nebezpečné**, musí sedět dynamický typ

dynamic_cast<typ>

- rozhoduje se za běhu programu
- jen pro ukazatele / reference na typy, které jsou součástí polymorfní hierarchie (tj. s alespoň jednou virtuální metodou)
 - k rozhodnutí používá ukazatel na virtuální tabulku
- ukazatel: pokud neuspěje, vrátí **nullptr**
- reference: pokud neuspěje, vyhodí výjimku **std::bad_cast**

Doporučení

- **dynamic_cast** používejte spíše opatrně
- preferujte vhodně navržené virtuální metody
- nadměrné používání **dynamic_cast** omezuje rozšiřitelnost objektové hierarchie
 - pokud přidání nové třídy do hierarchie znamená nutnost změnit metody ostatních třídy, je to typicky špatně
- jsou ovšem situace, kdy se **dynamic_cast** hodí
 - dotaz na implementaci rozhraní
 - dynamický výběr funkce podle více než jednoho parametru (ale existují i jiná a lepší řešení)

PŘÍKLAD: JEDNOTKY VE STRATEGICKÉ HŘE

- strategická hra s různými druhy jednotek
- chceme mít možnost zadat příkaz skupině jednotek
 - kontejner polymorfních objektů
- jednotky jsou různých druhů
 - některé se mohou pohybovat, některé mohou bojovat, ...

Řešení: jedno velké rozhraní

```
struct unit {  
    bool can_move() const = 0;  
    void move_to(position) = 0;  
    bool can_attack() const = 0;  
    void attack(unit&) = 0;  
    // ...  
};
```

- implementace budou často { **return false;** } a {}

Jiné řešení: rozhraní pro každou „schopnost“

```
struct unit { /* ... */ };
struct movable : unit { void move_to(position) = 0; };
struct attacker : unit { void attack(unit&) = 0; };

void move_all(std::vector<unit*>& units,
              position pos) {
    for (auto* u : units)
        if (auto* m = dynamic_cast<movable*>(u))
            m->move(pos);
}
```

- toto je vhodné použití **dynamic_cast**
- existují i jiná řešení (např. kompozice místo rozhraní)

PROČ NENÍ ČTVEREC PRAVOÚHELNÍK?

```
class square : public rectangle { /* ... */ }
```

- může porušovat substituční princip
- předpokládejme v `rectangle` metody `set_width`, `set_height`, `width`, `height`

```
void f(rectangle& r) {  
    r.set_width(5);  
    r.set_height(6);  
    assert(r.width() * r.height() == 30);  
}
```

- čtverec je pravoúhelník, ale objekt typu `square` není objektem typu `rectangle` – mají jiné *chování*

Hodnotové typy – (interní) stav

- nezajímá nás identita objektů, ale hodnota, kterou drží
- typicky se kopírují a přesouvají, typicky se používají „lokálně“
 - lokální proměnné, hodnotové položky
- příklad: `int`, `std::vector`, většina „našich“ typů do teď

Polymorfní typy – chování

- zajímá nás chování a identita objektů
- často se nekopírují ani nepřesouvají
 - když už, tak pomocí virtuálních metod (idiom *clone*)
- typicky dynamická alokace, „schovávání“ za ukazatele

Typy pro správu zdrojů – (externí) stav

- uvidíme příště