Dědičnost, polymorfismus, pozdní vazba

PB161 Programování v jazyce C++

Nikola Beneš

26. března 2024

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

Přístupová práva

access.cpp

Přístupnost položek vlastního typu

- **`public`**: smí přistupovat všichni (pro **`struct`** implicitní)
- **`private`**: smí přistupovat jen metody daného typu a *přátelé*
- **`protected`**: smí přistupovat 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
};
```

Přístupová práva

Přátelé

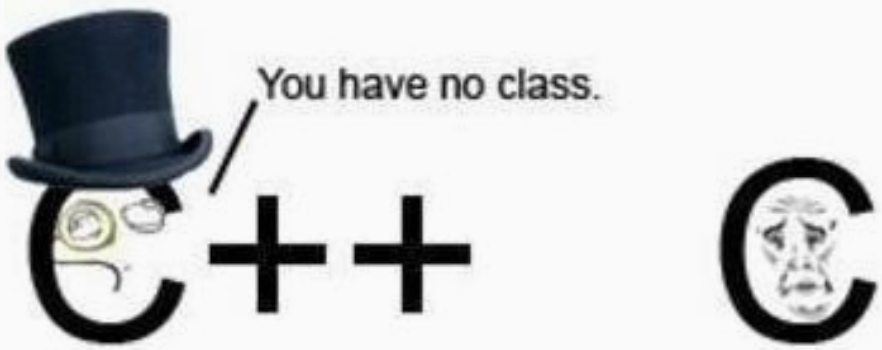
- přátelské funkce – už jsme viděli
- přátelské typy – `friend struct typ;`
  nebo jen `friend typ;` (pokud byl `typ` deklarován dříve)
- přátelské metody – `friend void typ::method(int);`

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)

Třídy



Klíčové slovo `class`

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

Polymorfismus

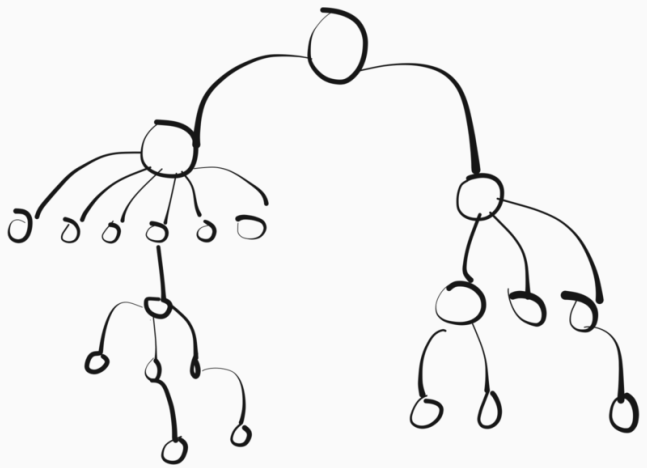
Princip polymorfismu

- 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)



Dědičnost

inheritance.cpp

```
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 vs. kompozice

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?

Příklady špatného použití dědičnosti

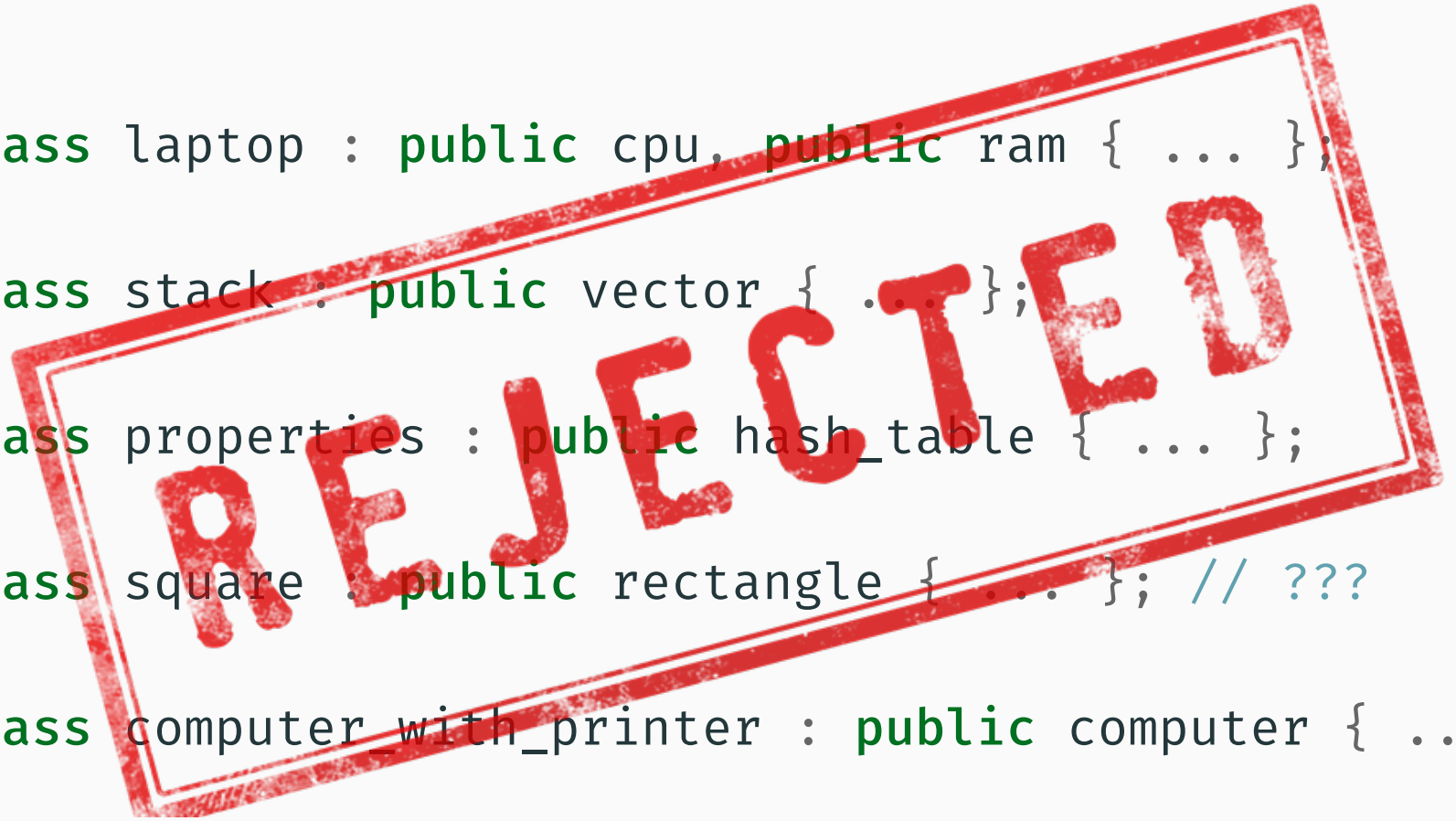
```
class laptop : public cpu, public ram { ... };

class stack : public vector { ... };

class properties : public hash_table { ... };

class square : public rectangle { ... }; // ???

class computer_with_printer : public computer { ... };
```



hiding.cpp

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 `::`

ctor_inheritance.cpp

Konstruktory

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

ctor_inheritance.cpp

Dědění konstruktorů

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

`dtor_inheritance.cpp`

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 destruktoru
  - destrukce položek v opačném pořadí deklarace
  - destrukce předka

Statický a dynamický typ

`static_type.cpp`

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;
```

Volání metod

Časná vazba (*early binding*)

- v C++ implicitní
- volá se metoda podle *statického typu* výrazu
  - tj. známá v době překladu
- 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

binding.cpp

Virtuální metody

- v předkovi označíme metodu slovem **`virtual`**
- metoda *se stejným typem včetně specifikátorů* 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

Virtuální metody a přístupová práva

`private_virtual.cpp`

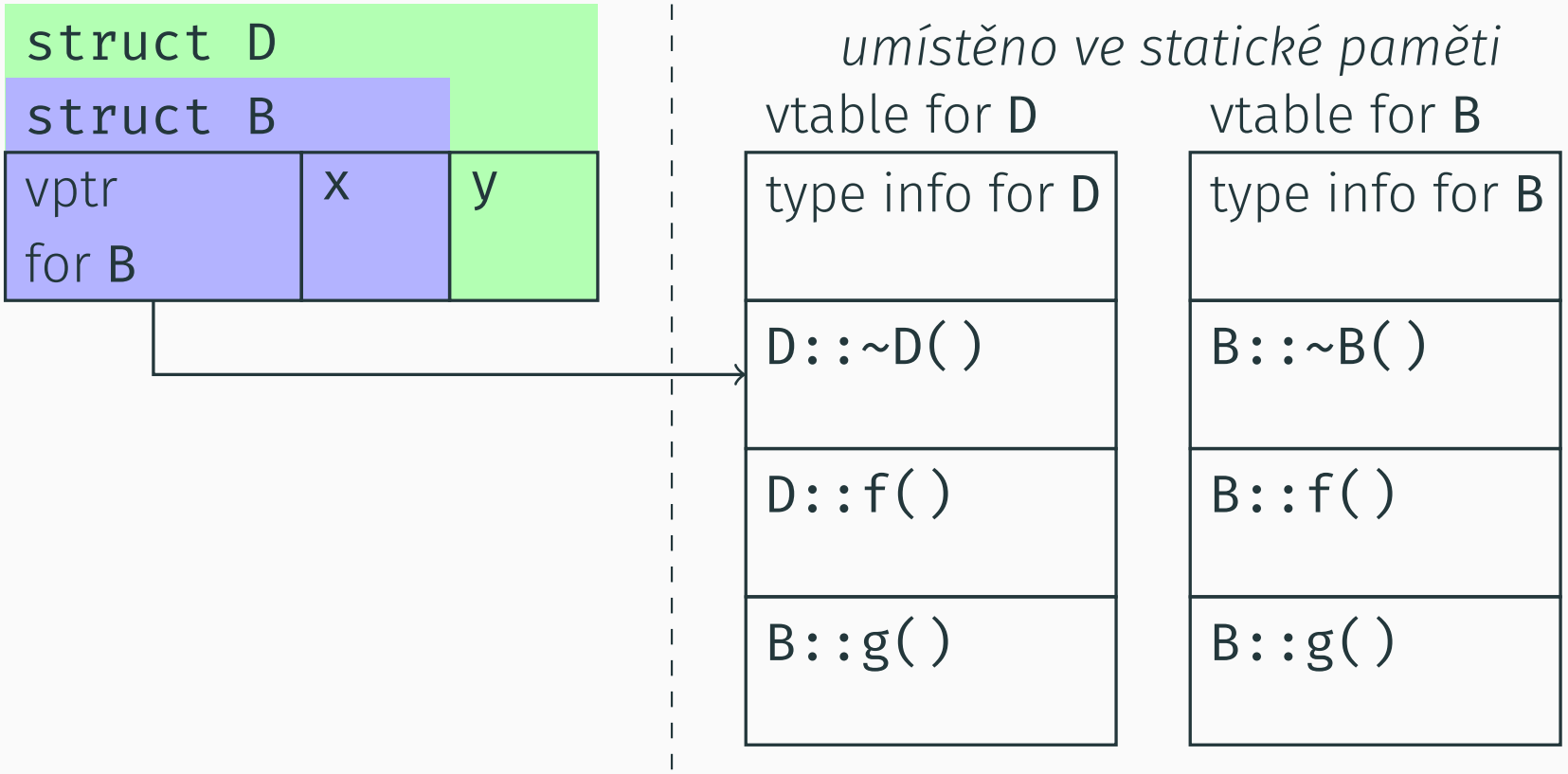
Vazba a přístupová práva jsou (v C++) nezávislé

- můžeme mít **`private`** nebo **`protected`** virtuální metody
- virtuální metody, která je **`public`** v předkovi, může být **`private`** v potomkovi apod.

- některá doporučení zahrnují používání **`private`** / **`protected`** virtuálních metod
  - implementační detail, který je možné změnit v potomkovi
  - http://www.gotw.ca/publications/mill18.htm
  - https://isocpp.org/wiki/faq/strange-inheritance#private-virtuals

Tabulka virtuálních metod (zjednodušeně)

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



Virtuální destruktor

virtual_dtor.cpp

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

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

init_destroy.cpp

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

Statický a dynamický typ

dynamic.cpp slicing.cpp

```
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: bázový podobjekt
  - tzv. *slicing* („ořezávání“ objektů)

Důsledek

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

Statický a dynamický typ – důsledky

dynamic.cpp

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`

Statický a dynamický typ – důsledky

dynamic.cpp

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`

virtual_copy.cpp

Kopírování

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

```
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);
    }
};
```

Statický a dynamický typ – důsledky

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

Abstraktní třídy, rozhraní

abstract.cpp

Čistě virtuální metoda

- metoda bez implementace v bázové třídě
- notace `virtual 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 destruktoru

Rozhraní (*interface*)

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

Vícenásobná dědičnost

- 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 bázové 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

dynamic.cpp

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 `{}`

Příklad: Jednotky ve strategické hře

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*

Různá typická využití tříd v C++

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

(inspirováno https://www.youtube.com/watch?v=_cwVvBsZE6M)