

PB161 Programování v jazyce C++
Přednáška 8
Dědičnost
Návrhové principy v OOP

Nikola Beneš

13. listopadu 2018

Dědičnost

Co už víte z minula?

Co už víte z minula?

- třída může dědit od jiné třídy
 - co všechno se dědí?

Co už víte z minula?

- třída může dědit od jiné třídy
 - co všechno se dědí?
 - atributy i metody
- dědičnost realizuje *podtypy*
 - vztah *IS-A*
 - lučištník je pozemní jednotka (an archer *is a* landunit)
 - pes je zvíře (a dog *is an* animal)
 - ukazatel/reference na předka se může odkazovat na potomka
- dědičnost může usnadnit znovupoužívání kódu
 - není to ale jediná možnost
 - kdy je to dobrý nápad?
- s dědičností souvisí pojmy časná a pozdní vazba

Dědičnost v C++

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

```
class Derived : public Base { /* ... */ };
```

- v tomto předmětu používáme pouze **public** dědičnost
- Derived má přístup k:

Dědičnost v C++

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

```
class Derived : public Base { /* ... */};
```

- v tomto předmětu používáme pouze **public** dědičnost
- Derived má přístup k:
 - veřejným a protected atributům a metodám Base
- uživatelé Derived mají přístup k:

Dědičnost v C++

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

```
class Derived : public Base { /* ... */};
```

- v tomto předmětu používáme pouze **public** dědičnost
- Derived má přístup k:
 - veřejným a protected atributům a metodám Base
- uživatelé Derived mají přístup k:
 - veřejným atributům a metodám Derived včetně zděděných veřejných atributů a metod
- v jakém pořadí se volají konstruktory?

Dědičnost v C++

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

```
class Derived : public Base { /* ... */};
```

- v tomto předmětu používáme pouze **public** dědičnost
- Derived má přístup k:
 - veřejným a protected atributům a metodám Base
- uživatelé Derived mají přístup k:
 - veřejným atributům a metodám Derived včetně zděděných veřejných atributů a metod
- v jakém pořadí se volají konstruktory?
 - konstruktor předka se provede před konstruktorem potomka
 - předek je *jakoby* prvním atributem potomka
- v jakém pořadí se volají destruktory?

Dědičnost v C++

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

```
class Derived : public Base { /* ... */};
```

- v tomto předmětu používáme pouze **public** dědičnost
- Derived má přístup k:
 - veřejným a protected atributům a metodám Base
- uživatelé Derived mají přístup k:
 - veřejným atributům a metodám Derived včetně zděděných veřejných atributů a metod
- v jakém pořadí se volají konstruktory?
 - konstruktor předka se provede před konstruktorem potomka
 - předek je *jakoby* prvním atributem potomka
- v jakém pořadí se volají destruktory?
 - destruktory potomka se provede před destruktorem předka

Mám-li referenci/ukazatel na objekt a zavolám na něm metodu, která metoda se zavolá?

Mám-li referenci/ukazatel na objekt a zavolám na něm metodu, která metoda se zavolá?

- časná vazba (early binding): podle typu ukazatele/reference
- pozdní vazba (late binding): podle skutečného typu objektu

Který typ vazby je v C++ implicitní a proč?

Mám-li referenci/ukazatel na objekt a zavolám na něm metodu, která metoda se zavolá?

- časná vazba (early binding): podle typu ukazatele/reference
- pozdní vazba (late binding): podle skutečného typu objektu

Který typ vazby je v C++ implicitní a proč?

Vynucení pozdní vazby – klíčové slovo `virtual`

- metoda s klíčovým slovem `virtual` ve třídě předka
- překrývající metody ve třídách potomků mohou a nemusí mít klíčové slovo `virtual` (metoda je virtuální každopádně)
- od C++11: klíčové slovo `override`
 - sdělujeme úmysl překrýt (override) metodu z předka
 - překladač nás hlídá

V čem je problém?

```
class Base {
public:
    ~Base() { std::cout << "~Base()\n"; }
};
class Derived : public Base {
public:
    ~Derived() { std::cout << "~Derived()\n"; }
};
int main() {
    Derived d;
    std::unique_ptr<Base> uptr = std::make_unique<Derived>();
}
```

lecture08_03.cpp, lecture08_04.cpp

Časná a pozdní vazba – Destruktory

V čem je problém?

```
class Base {
public:
    ~Base() { std::cout << "~Base()\n"; }
};
class Derived : public Base {
public:
    ~Derived() { std::cout << "~Derived()\n"; }
};
int main() {
    Derived d;
    std::unique_ptr<Base> uptr = std::make_unique<Derived>();
}
```

Co s tím?

lecture08_03.cpp, lecture08_04.cpp

Časná a pozdní vazba – Destruktory

V čem je problém?

```
class Base {
public:
    ~Base() { std::cout << "~Base()\n"; }
};
class Derived : public Base {
public:
    ~Derived() { std::cout << "~Derived()\n"; }
};
int main() {
    Derived d;
    std::unique_ptr<Base> uptr = std::make_unique<Derived>();
}
```

Co s tím? A co když Base ani Derived nemají destruktory?

[lecture08_03.cpp](#), [lecture08_04.cpp](#)

Herb Sutter: A *base class destructor* should be either public and virtual, or protected and nonvirtual.

In brief, then, you're left with one of two situations. Either:

- 1 *you want to allow polymorphic deletion through a base pointer, in which case the destructor must be virtual and public; or*
- 2 *you don't, in which case the destructor should be nonvirtual and protected, the latter to prevent the unwanted usage.*

Kdy psát `virtual`?

- je daná třída součástí objektové hierarchie?
- bude se od dané třídy dědit?
- je metoda určená k tomu, aby ji potomci překrývali vlastní implementací?

Kdy psát `virtual`?

- je daná třída součástí objektové hierarchie?
- bude se od dané třídy dědit?
- je metoda určená k tomu, aby ji potomci překrývali vlastní implementací?

Jak psát virtuální metody?

- v předkovi použijte klíčové slovo `virtual`
- v potomcích použijte klíčové slovo `override`
 - ušetří to spoustu času stráveného hledáním chyb
 - v potomcích není potřeba psát `virtual`
- co když nemám v předkovi žádnou vhodnou implementaci?

Kdy psát `virtual`?

- je daná třída součástí objektové hierarchie?
- bude se od dané třídy dědit?
- je metoda určená k tomu, aby ji potomci překrývali vlastní implementací?

Jak psát virtuální metody?

- v předkovi použijte klíčové slovo `virtual`
- v potomcích použijte klíčové slovo `override`
 - ušetří to spoustu času stráveného hledáním chyb
 - v potomcích není potřeba psát `virtual`
- co když nemám v předkovi žádnou vhodnou implementaci?
 - použijte *čistě virtuální* metodu (syntax = 0)

Abstraktní třída

- má alespoň jednu čistě virtuální metodu
- nelze od ní vytvářet instance

Čistě abstraktní třída

- všechny její metody jsou čistě virtuální (s případnou výjimkou destruktoru)

Rozhraní (interface)

- čistě abstraktní třída bez atributů (s případnou výjimkou statických položek)
- neдрží žádná data, deklaruje pouze, které metody bude možno volat
- v C++ není speciální klíčové slovo, jen dohoda

Vícenásobná dědičnost

Je možno dědit od více tříd?

Vícenásobná dědičnost

Je možno dědit od více tříd?

- ano

Je to dobrý nápad?

Vícenásobná dědičnost

Je možno dědit od více tříd?

- ano

Je to dobrý nápad?

- jak kdy; bez problému, pokud dědíme pouze od rozhraní

```
class IPrinter { /* ... */ };  
class IScanner { /* ... */ };  
class Copier: public IPrinter, public IScanner { /* ... */ };
```

Co když dědíme od jiných druhů tříd?

Vícenásobná dědičnost

Je možno dědit od více tříd?

- ano

Je to dobrý nápad?

- jak kdy; bez problému, pokud dědíme pouze od rozhraní

```
class IPrinter { /* ... */ };  
class IScanner { /* ... */ };  
class Copier: public IPrinter, public IScanner { /* ... */ };
```

Co když dědíme od jiných druhů tříd?

- může nastat problém, pokud se tyto třídy nějakým způsobem překrývají (mají stejnou metodu/atribut, dědí od stejné třídy)
- problém s děděním od stejné třídy se nazývá *diamond problem*
- řeší se virtuální dědičností (pokročilejší téma, v následujícím předmětu)

lecture08_06.cpp

Co už víte o přetypování v C++

Co už víte o přetypování v C++

- Céčkové přetypování ve stylu `(typ)hodnota` není typově bezpečné a nemělo by se v C++ používat; proč?
 - tohle přetypování postupně zkouší všechny možnosti (včetně `reinterpret_cast` a `const_cast`)
 - taky není syntakticky moc pěkné (jakou má prioritu?)
- pro přetypování mezi primitivními typy můžeme použít `static_cast<typ>(hodnota)`
 - změna celočíselné hodnoty na `enum`
 - změna ukazatele na `void *` a naopak (pozor na nedef. chování)

http://en.cppreference.com/w/cpp/language/explicit_cast

Přetypování v objektové hierarchii

- ukazatel na potomka se umí implicitně konvertovat na ukazatel na předka (podobně pro reference)
- opačný směr?

Přetypování v objektové hierarchii

- ukazatel na potomka se umí implicitně konvertovat na ukazatel na předka (podobně pro reference)
- opačný směr?

- `static_cast<typ>(hodnota)`
 - jen když zcela určitě vím, že objekt je skutečně daného typu
 - co když to nevím? nedefinované chování!
- `dynamic_cast<typ>(hodnota)`
 - rozhoduje se *za běhu* programu
 - jen pokud je někde v hierarchii virtuální metoda (proč?)
 - používejte zřídka a jen pokud k tomu máte dobrý důvod, preferujte rozumně navržené virtuální metody

Dědičnost a přetypování

- `dynamic_cast`<typ>(hodnota)
 - k určení, zda je hodnota skutečně daného typu, se používá ukazatel na virtuální tabulku
- pokud `dynamic_cast` neuspěje:
 - pokud je typ ukazatel, vrátí `nullptr`
 - pokud je typ reference, vyhodí výjimku typu `std::bad_cast`

Dědičnost je vztah *IS-A*

- potomek má vlastnosti předka
- potomka je možno přetypovat na předka

Kompozice je vztah *HAS-A*

- objekt se skládá z jiných objektů
 - auto se skládá z motoru, kol, dveří, ...
 - počítač se skládá z procesoru, paměti, disků, ...
- reprezentace v OOP jazyce:
 - třída má atributy, které jsou typu jiné třídy
- třída tím obsahuje vlastnosti těchto jiných tříd
- není ale jejich potomkem, nemůže je zastoupit

Proč o tom mluvíme? Časté zneužití dědičnosti tam, kde by byla lepší kompozice.

Příklady špatného použití dědičnosti

```
class Laptop : public CPU, public RAM { /* ... */ };
```

```
class Stack : public Vector { /* ... */ };
```

```
class Properties : public HashTable { /* ... */ };
```

```
class Square : public Rectangle { /* ... */ }; // ???
```

```
class ComputerWithPrinter : public Computer { /* ... */ };
```


Příklady špatného použití dědičnosti

```
class Laptop : public CPU, public RAM { /* ... */ };
```

```
class Stack : public Vector { /* ... */ };
```

```
class Properties : public HashTable { /* ... */ };
```

```
class Square : public Rectangle { /* ... */ }; // ???
```

```
class ComputerWithPrinter : public Computer { /* ... */ };
```

Složitější:

```
class Teacher : public Person { /* ... */ };
```

```
class Student : public Person { /* ... */ };
```

- co když je někdo zároveň student a učitel?
- kompozice místo dědičnosti: role

Návrhové principy a vzory

Jak správně navrhnout objektovou hierarchii?

- co umístit do jedné třídy?
- jaké vztahy mezi třídami?

Jak se pozná špatný návrh?

- malé změny vyžadují velké úpravy
- změny způsobí nečekané problémy
- velká provázanost (s konkrétní aplikací, s jiným kódem)

Základní návrhové principy pro OOP

- **SRP** – *The Single Responsibility Principle*
- **OCP** – *The Open/Closed Principle*
- **LSP** – *The Liskov Substitution Principle*
- **ISP** – *The Interface Segregation Principle*
- **DIP** – *The Dependency Inversion Principle*

- [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))
(odkazy v části *References*)

The Single Responsibility Principle

- (už jsme viděli)
- „Třída by měla mít jediný důvod ke změně.“
- obecnější princip: každý se stará o jednu věc

Důsledky porušení

- změna jedné z funkcionalit vyžaduje rekompilaci
- úprava kódu poruší více funkcionalit

Příklad porušení SRP

- třída `Rectangle` zároveň vykresluje čtverec a počítá jeho obsah
- někteří uživatelé potřebují obě tyto funkcionality, někteří jen jednu

Lepší řešení

- rozdělit funkcionality `Rectangle` do dvou tříd
- o počítání obsahu se postará `GeometricRectangle`

The Open/Closed Principle

- „Třídy by mělo být možné rozšiřovat, ale bez jejich modifikace.“
- nejen třídy (moduly, jiné entity)
- *open for extension*: možnost přidávat chování
- *closed for modification*: možnost použití jiným kódem

- dosahuje se použitím dědičnosti a abstrakce

Související

- nepoužívat globální proměnné
- soukromé atributy objektů
- používat `dynamic_cast` opatrně (proč?)

Důsledek porušení: kaskáda změn v kódu

The Liskov Substitution Principle

- (už jsme v podstatě viděli)
- „Potomek může zastoupit předka.“
- funkce očekávající objekt typu předka musí fungovat s objekty typu potomka, aniž by o tom věděly
- úzce souvisí s OCP

Důsledek porušení: neočekávané chování (špatné předpoklady)

Příklad porušení LSP

- mějme třídu Rectangle a třídu Square, která z ní dědí
- obě mají metody setWidth a setHeight (a odpovídající get)

```
void f(Rectangle& r) {  
    r.setWidth(5);  
    r.setHeight(6);  
    assert(r.getWidth() * r.getHeight() == 30);  
}
```

- kde je problém?

Příklad porušení LSP

- mějme třídu `Rectangle` a třídu `Square`, která z ní dědí
- obě mají metody `setWidth` a `setHeight` (a odpovídající `get`)

```
void f(Rectangle& r) {  
    r.setWidth(5);  
    r.setHeight(6);  
    assert(r.getWidth() * r.getHeight() == 30);  
}
```

- kde je problém?
- čtverec je pravoúhelník, ale objekt typu `Square` není objektem typu `Rectangle` – mají jiné *chování*

The Interface Segregation Principle

- „Vytvářejte specifická rozhraní pro specifické klienty.“
- více malých rozhraní je lepší než jedno obrovské
- klienti by neměli být nuceni záviset na rozhraních, které nepoužívají
- souvisí s SRP

Jak dosáhnout?

- rozhraní s mnoha metodami: kdo je používá?
- opravdu používají všichni všechny metody?
- rozdělit metody do samostatných rozhraní
- použití vícenásobné dědičnosti

The Dependency Inversion Principle

- obecné třídy by neměly záviset na specializovaných třídách; všichni by měli záviset na abstrakcích
- abstrakce by neměly záviset na detailech, ale naopak
- souvisí s OCP a LSP

- použití rozhraní (čistě abstraktních tříd)

Důsledky porušení:

- přidání podpory nového typu vede ke změně v obecné třídě
- obecná třída použitelná jen pro co byla implementována

Příklad porušení:

- třída Worker s metodou work() a třída Manager, která ji používá
- závislost Manager → Worker
- co když budeme chtít přidat novou třídu SuperWorker?

Řešení:

- vytvoření abstrakce (rozhraní) IWorker
- závislosti Manager → IWorker, Worker → IWorker, ...

Keep It Simple, Stupid!

Princip KISS

- jednoduché, postačující řešení může být lepší než komplikované a rafinované
 - snazší pochopení (dalšími vývojáři, námi samotnými)
 - menší riziko chyby
- kompromis mezi aktuálními a budoucími požadavky
 - příliš mnoho budoucích požadavků návrh komplikuje
 - omezení na aktuální požadavky vadí budoucí rozšiřitelnosti
- průběžný vývoj, refaktorizace (nebát se!)

Motivace

- opakující se programátorské problémy
- opakující se způsoby řešení
- vhodné konstrukce pro řešení: návrhové vzory

- kniha *Design Patterns: Elements of Reusable Object-Oriented Software* (23 vzorů)
- autoři *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* (tzv. Gang of Four, GoF)

- existují i jiné návrhové vzory
- kde začít? https://en.wikipedia.org/wiki/Software_design_pattern

Co je návrhový vzor?

- (jednoduchý) způsob, jak řešit skupiny podobných problémů
- opakovatelný, (více méně) nezávislý na konkrétní aplikaci či jazyce

Pozitiva

- řeší běžné problémy
- mohou zlepšovat kód a jeho udržovatelnost

Kritika

- zbytečně složité na jednoduché problémy
- nadužívání může vést ke komplikovanějšímu kódu
- často řeší jen nedokonalost používaného jazyka

<https://kahoot.it>

<https://kahoot.it>

```
class A {
public:
    void f() { std::cout << "Af "; g(); }
    virtual void g() { std::cout << "Ag "; }
};

class B : public A {
public:
    void f() { std::cout << "Bf "; }
    void g() override { std::cout << "Bg "; }
};

int main() {
    B b; A* ptr = &b; ptr->f();
}
```