

PB161 — 3. přednáška (5. října 2015)

Organizační

Vnitrosemestrální test

- bude se konat 2. 11. 2015 v době přednášky
- dvě skupiny (12-13, 13-14)
- papírový odpovědník, max. 20 bodů
- bude obsahovat náplň předchozích přednášek
 - co udělá zadaný kód, zkompile se apod.

Reference, const

Motivace

- práce s ukazateli (vzpomeňte si na C) může být nebezpečná
- ukazatel může být neinicializovaný
- ukazatel může být NULL (v C++11 nullptr)
 - odbočka: proč je nullptr lepší než NULL?
 - NULL (převzato z C) je definováno jako číslo 0
 - nullptr je speciální objekt typu ukazatel
 - rozdíl bude vidět u přetěžování funkcí

Datový typ reference

- alternativní jméno pro objekt/proměnnou (alias)
- značí se symbolem &

```
int a = 1;
int b = 2;
int & refA = a; // refA je reference na a
int & refA2 = a; // refA2 je také reference na a
int & ref;      // chyba!
a      += 9;
refA   += 17;
refA2 += 15;
// teď je v proměnné a hodnota 42
```

- jednodušší než ukazatel, ale bezpečnější

Rozdíly mezi ukazateli a referencemi

- ukazatel může být neinicializovaný, reference musí být vždy inicializovaná

```
int * ptr; // OK
int & ref; // chyba
```

- ukazatel může být nullptr, reference vždy ukazuje na skutečný objekt (i když ten objekt nemusí být validní, o tom později)

```
int * ptr = nullptr; // V C nebo C++03: int * ptr = NULL;
// s referencemi nic takového nejde
```

- ukazatel se může měnit (může ukazovat na něco jiného), reference je napevno, není způsob jak „přesměrovat“ referenci

```
int a = 1;
int b = 2;
```

```
int * ptr = &a; // ptr ukazuje na a
ptr = &b;      // teď ptr ukazuje na b
```

```
int & ref = a; // ref je reference na a
ref = b;      // do proměnné a se teď vložila hodnota 2
ref = &b;     // chyba! do proměnné typu int
              // se snažíme přiřadit ukazatel
```

- není možno se přímo odkazovat na proměnnou typu reference, tato proměnná skutečně funguje jako alias

```
int * ptr = &a;
fun1(ptr); // funkci fun1 se předal ukazatel ptr
int & ref = a;
fun2(ref); // funkci fun2 se předala proměnná a
fun2(a);   // stejný efekt jako předchozí řádek
```

Použití referencí

- bezpečnější varianta ukazatelů, pokud nepotřebujeme ukazatel přesměrovávat
- předávání parametrů (tzv. volání odkazem) funkcím a metodám

Předávání parametrů referencí

- umožňuje funkci měnit hodnotu proměnné
 - volání funkce ale vypadá stejně jako volání hodnotou
 - změna argumentu uvnitř funkce se projeví navenek
- v jazyce C se řešilo pomocí ukazatelů
 - méně bezpečné (NULL, neinicializovaný ukazatel)
 - vyžaduje v místě volání psát dereferenci
 - všimněte si rozdílů:

```
// jazyk C++
void swap(int & x, int & y) {
```

```

    int temp = x;
    x = y;
    y = temp;
}

int a = 3;
int b = 7;

swap(a, b);

/* jazyk C */
void swap(int * x, int * y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int a = 3;
int b = 7;

swap(&a, &b);

```

- samozřejmě předávání ukazatelem (styl C) funguje i v C++, ale předávání referencí je lepší (bezpečnější, čitelnější)

Konstantní reference

- klíčové slovo `const` stejně jako v C
- dá se použít i s referencemi

```

int a = 3;
const int & refA = a;

cout << refA; // OK, vypíše 3
a    = 7;      // OK, hodnota a je 7
refA = 17;     // chyba, refA je konstantní

```

- předávání parametrů konstantní referencí
 - ukazuje záměr neměnit proměnnou (hlídá překladač)
 - hlavní využití: předávání větších objektů

```

void comparePersons(const Person & p1, const Person & p2) {
    cout << p1.name << " is ";
    if (p1.age < p2.age)
        cout << "younger than";
    else if (p1.age > p2.age)
        cout << "older than";
}

```

```

    else
        cout << "the same age as";
        cout << p2.name << endl;
}

void f1(BigObject x)          { /* ... */ }
void f2(const BigObject & x) { /* ... */ }

```

```
BigObject y;
```

```

f1(y); // vytvoří kopii objektu y
f2(y); // nevytváří kopii, předává konstantní referenci

```

- **const** se dá chápat jako *read-only* (**readonly** bylo klíčové slovo v původním Stroustrupově návrhu C with Classes)
- Situace s předáváním hodnotou vs. předáváním konstantní referencí je v moderní době trochu složitější: moderní překladače umí techniky jako *copy elision*, *return value optimization* apod., které mnohá použití volání hodnotou optimalizují.

Reference a l-hodnoty/p-hodnoty

- l-hodnota (l-value) je něco, co může stát na levé straně přiřazení
 - někde to „bydlí“, má to adresu
 - např. proměnná
- p-hodnota (r-value) je něco, co může stát na pravé straně přiřazení
 - např. číselná hodnota, dočasný objekt
- konstantní reference „chytá“ všechno, nekonstantní „chytá“ jen l-hodnoty
 - konstantní reference má proto nižší prioritu u přetěžování (viz níže)
- v C++11 je situace opět složitější (r-value reference, move semantics) – více na speciální přednášce o C++11 koncem semestru

```

void lfun(int & x);
void rfun(const int & x);

int a = 7;
lfun(a); // OK
lfun(17); // chyba! 17 není l-hodnota

rfun(a); // OK
rfun(17); // OK

```

Reference jako návratová hodnota

- reference je možno i vracet, jsou to l-hodnoty (pokud nejsou konstantní)

```

int & fun(int & x) {
    x += 17;
}

```

```

    return x;
}

```

```
int y = 10;
```

```
fun(y) = 25;
```

// v tuto chvíli je v y hodnota 25

Reference jako atributy

- atributem třídy může být i reference
- jak inicializovat?

// takhle to nejde:

```

class Dog {
    Person & owner;
public:
    Dog(Person & o) {
        owner = o;  // CHYBA!
    }
};

```

// je třeba použít inicializační sekci konstruktoru:

```

class Dog {
    Person & owner;
public:
    Dog(Person & o) : owner(o) {}
};

```

- toto je jeden z důvodů existence inicializační sekce

Konstatní metody

- na úlohách pro první a druhé cvičení jste si jistě všimli u některých metod klíčového slova `const` za seznamem parametrů

```

class Person {
    std::string name;
    int age;
public:
    void setAge(int a) {
        age = a;
    }
    int getAge() const {
        return age;
    }
};

```

- konstantní metody se zavazují neměnit vnitřní stav objektu (hlídá překladač)
- úzce souvisí s předáváním objektů konstantní referencí (a samozřejmě i konstantním ukazatelem)

```
void funR(const Person & person) {
    cout << person.getAge(); // OK
    person.setAge(18);       // chyba!
}

void funP(const Person * person) {
    if (person) {
        cout << person->getAge(); // OK
        person->setAge(42);        // chyba!
    }
}
```

- rozumný styl programování: označovat jako **const** všechny metody, které nehodlají měnit vnitřní stav objektu
 - vyšší bezpečnost
 - nemůžeme omylem změnit stav objektu, překladač zahlásí chybu
- obecně (snad znáte už z C): používat **const** všude, kde je to možné

Přetěžování funkcí

Přetěžování (overloading)

- už jsme vlastně viděli minule u konstruktorů
- možnost vytvořit několik různých funkcí/metod se stejným jménem, ale různou implementací
- musí se lišit typem nebo počtem parametrů

```
void f(int x) {
    cout << "f s parametrem int: " << x << endl;
}

void f() {
    cout << "f bez parametrů" << endl;
}

void f(double d) {
    cout << "f s parametrem double: " << d << endl;
}

f(3); // zavolá void f(int)
f(5.0); // zavolá void f(double)
f(); // zavolá void f()
```

- pozor! nestačí, pokud se liší pouze návratovým typem; proč?

```
void f(int x) { /* ... */ }
```

```
int f(int x) { /* ... */ } // CHYBA!
```

- stejně jako funkce se mohou přetěžovat i metody tříd
- metody se mohou lišit i jen specifikátorem const (uvidíme příště u přetíženého operátoru [])

```
class File {
// ...
public:
    void write(int num);
    void write(std::string str);
// ...
};
```

- to, která funkce/metoda se má zavolat, se rozhoduje při překladu podle typů skutečných parametrů

Pro zvědavé: Jak to ve skutečnosti funguje?

- překladač C++ mění jména funkcí, přidává k nim typy parametrů
 - tzv. *name mangling*
 - není žádný standard, závisí na konkrétním překladači
- příklad (gcc): jméno funkce `void fun(int, char)` se interně změní na `_Z3funic`, jméno funkce `int fun(int &)` se změní na `_Z3funRi`

Přetěžování a reference

- pokud máme funkci přetíženou pro nekonstantní a konstantní referenci, nekonstantní má přednost

```
void f(int &);
```

```
void f(const int &);
```

```
int x;
```

```
f(x); // zavolá se první funkce
```

```
f(5); // zavolá se druhá funkce
```

```
int & ref = x;
```

```
const int & cref = x;
```

```
f(ref); // první
```

```
f(cref); // druhá
```

- mít zároveň funkci přetíženou pro referenci a pro základní typ není vhodné; překladač si bude stěžovat na nejednoznačnost

```

void f(int &);
void f(int);

int x;
f(x); // chyba při překladu
f(5); // OK, zavolá se druhá funkce

```

Přetěžování a NULL

- podívejme se, co se stane v C++03, pokud budeme mít funkci přetíženou pro `int` a ukazatel

```

void f(int x);
void f(char * s);

f(NULL); // zavolá se PRVNÍ funkce, NULL je definováno jako 0

```

- proto máme v C++11 speciální ukazatel `nullptr`

```

f(nullptr); // OK, zavolá se druhá funkce

```

- NULL může být v C++11 definováno různě, proto je lépe jej nepoužívat a zvyknout si na používání `nullptr`

Implicitní parametry

- funkce a metody v C++ (včetně konstruktorů) mohou mít parametry s implicitní hodnotou
 - implicitní parametry musí být nejvíce vpravo v seznamu parametrů

```

void f(int x, int y = 10, int z = 20);

```

```

f(3, 4, 5);
f(3, 4);    // zavolá se f(3, 4, 20);
f(3);      // zavolá se f(3, 10, 20);

```

```

void g(int x = 10, int y); // chyba!

```

- při oddělení deklarace a definice se implicitní hodnoty píšou do deklarace, do definice už ne
 - někde bývá zvykem připomenout implicitní hodnotu v komentáři

```

// v deklaraci není nutné uvádět názvy parametrů
void f(int = 10, int = 20);

void f(int x /* = 10 */, int y /* = 20 */) {
    // ...
}

```


Statické atributy a metody

- položky tříd (atributy a metody), které jsme dosud viděli, patřily objektům (instancím třídy)
- statické položky třídy: uvozeny klíčovým slovem **static**
 - patří třídě samotné; nepatří žádnému objektu
 - objekty k nim mohou přistupovat

```
class Person {
    std::string name;
    int age;
    static int count;
public:
    Person(std::string n, int a) : name(n), age(a) {
        ++count;
    }
    // ...
    static int getCount {
        return count;
    }
};

// inicializace statických atributů mimo deklaraci třídy
int Person::count = 0;

Person franta("Franta", 20);
Person pepa("Pepa", 21);
Person jimmy("James Bond", 25);

// volání statické metody, bez konkrétního objektu
cout << Person::getCount() << endl; // vypíše 3
```

- příklad použití statických atributů: automatické přidělování ID

```
class Thing {
    int id;
    static int id_generator;
public:
    Thing() : id(id_generator++) {}
    int getID() const {
        return id;
    }
};

int Thing::id_generator = 0;
```

```

Thing t1;
Thing t2;
Thing t3;

t3.getID(); // vrátí 2

```

Dědičnost a kompozice

- minule jsme se začali bavit o dědičnosti v OOP a její realizaci v C++, teď se o ní budeme bavit podrobněji

Motivace pro dědičnost: Rozhraní

- chceme psát kód, který využívá tiskárnu
- budeme psát speciální kód pro každý typ tiskárny?
- chceme využívat rozhraní (interface)
- při návrhu rozhraní chceme zachytit, co všechno musí umět třída, aby se mohla vydávat za příslušníka dané skupiny
 - např. tiskárna musí umět přijmout dokument k tisku a vypsát aktuálně tisknuté dokumenty
 - za tiskárnu se ale klidně může vydávat i něco jiného (PDF writer)

```

// interface (čistě abstraktní třída bez atributů)
class IPrinter {
public:
    virtual std::string GetPendingDocuments() const = 0;
    virtual void PrintDocument(const string & document) = 0;
    virtual ~IPrinter() {}
};

// konkrétní třída
class HPLaserJet : public IPrinter {
// ...
public:
    HPLaserJet();
    std::string GetPendingDocuments() const;
    void PrintDocument(const string & document);
    ~HPLaserJet();
};

// ...

void HPLaserJet::PrintDocument(const string & document) {
    // ...
}

```

- můžeme psát kód využívající rozhraní IPrinter, aniž bychom něco tušili o tom, jakou konkrétní tiskárnou cílový systém disponuje

```

IPrinter * printer = new HPLaserJet();
printer->PrintDocument("top_secret.txt");
// ...
delete printer;

```

- pozn. o operátorech **new** a **delete**, které zajišťují dynamickou alokaci v C++, si řekneme příště

Motivace pro dědičnost: Hierarchie abstrakce

- inkoustové a laserové tiskárny se od sebe liší, uvnitř těchto skupin je ale mnoho vlastností společných

```

class InkPrinter : public IPrinter {
// ...
};

class LaserPrinter : public IPrinter {
// ...
};

class HPLaserJet : public LaserPrinter {
// ...
};

```

Dědičnost

- nástroj pro podporu abstrakce
- potomci dodržují rozhraní definované předkem, mohou ale měnit chování (implementaci)
- omezuje duplicity v kódu

Dědičnost v C++

- třída může dědit od jedné nebo více tříd
- specifikátory přístupových práv
 - **public**: zděděné položky dědí práva od předka
 - **protected**: veřejné zděděné položky se mění na **protected**
 - **private**: všechny zděděné položky se mění na **private**
 - bez specifikátoru: u **class** jako **private**, u **struct** jako **public**
 - **virtual**: lze kombinovat s jedním z předchozích, řeší problémy s vícenásobnou dědičností (viz později)

```

class A {
public: // veřejné, vidí všichni
    int x; // normálně se pro atributy nepoužívá
           // zde jen pro ilustraci
protected: // vidí potomci

```

```

    int y;
private: // soukromé, vidí jen instance třídy A
    int z;
};

class B : public A {
    // x je public
    // y je protected
    // z zde není přístupné
};

class C : protected A {
    // x je protected
    // y je protected
    // z zde není přístupné
};

class D : private A {
    // x je private
    // y je private
    // z zde není přístupné
};

class E : public D {
    // žádné z x, y, z zde není přístupné
};

```

- nejběžnější specifikátor je **public**, ostatní typy dědičnosti mají omezené použití a v tomto předmětu se s nimi nesetkáte (nicméně je dobré o nic vědět, kdybyste je náhodou někdy potkali „tam venku“)

Dědičnost a reference

- z minula už víme, že ukazateli na předka můžeme předat potomka
- podobně to funguje i s referencemi

```

class Animal {
    // ...
};

class Dog : public Animal {
    // ...
};

class Person {
    // ...
    void playWith(const Animal & animal);
}

```

```
// ...
};

// můžeme psát

Person tommy;
Dog lassie;

tommy.playWith(lassie);
```

Časná a pozdní vazba

- v C++ je implicitní časná vazba: metoda, která se má zavolat, je určena staticky (při kompilaci)

```
class Animal {
public:
    void makeSound() const {
        std::cout << "<generic animal sound>\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() const {
        std::cout << "Whoof!\n";
    }
};

Dog lassie;
Animal & refAnimal = lassie;
Animal * ptrAnimal = &lassie;

lassie.makeSound(); // "Whoof!"
refAnimal.makeSound(); // "<generic animal sound>"
ptrAnimal->makeSound(); // "<generic animal sound>"
```

- můžeme přikázat pozdní vazbu: metoda, která se má zavolat, je pak určena dynamicky (za běhu programu)

```
class Animal {
public:
    virtual void makeSound() const { // metoda makeSound je virtuální
        std::cout << "<generic animal sound>\n";
    }
};
```

```

class Dog : public Animal {
public:
    // zde může i nemusí být slovo virtual
    // metoda makeSound je každopádně stále virtuální
    // (není způsob, jak přestat být virtuální)
    void makeSound() const {
        std::cout << "Whoof!\n";
    }
};

```

```

Dog lassie;
Animal & refAnimal = lassie;
Animal * ptrAnimal = &lassie;

```

```

lassie.makeSound();    // "Whoof!"
refAnimal.makeSound(); // "Whoof!"
ptrAnimal->makeSound(); // "Whoof!"

```

- pozor při předefinování metod: typ předefinované metody musí být úplně stejný, včetně případného const

```

class Animal {
public:
    virtual void makeSound() const {
        std::cout << "<generic animal sound>\n";
    }
};

```

```

class Dog : public Animal {
public:
    // zde jsme vytvořili NOVOU virtuální metodu
    // která skrývá metodu void Dog::makeSound() const
    virtual void makeSound() {
        std::cout << "Whoof!\n";
    }
};

```

```

Dog lassie;
Animal & refAnimal = lassie;
Animal * ptrAnimal = &lassie;

```

```

lassie.makeSound();    // "Whoof!"
refAnimal.makeSound(); // "<generic animal sound>"
ptrAnimal->makeSound(); // "<generic animal sound>"

```

- rozdíl mezi časnou a pozdní vazbou z hlediska rychlosti

- časná vazba je statická a proto rychlejší
- pozdní vazba vyžaduje nahlédnutí do tabulky virtuálních funkcí a je proto pomalejší

Přetěžování, předefinování, skrývání (overloading, overriding, hiding)

- přetěžování je deklarace několika metod se stejným názvem a různými parametry
- předefinování je nahrazení definice virtuální metody ve třídě předka novou definicí ve třídě potomka
- ke skrývání dochází, pokud je v potomkovi deklarovaná metoda stejného jména jako v předkovi (a buď není virtuální nebo má jiné parametry, případně se liší kvalifikátorem const)

```
class Account {
// ...
public:
    virtual void deposit(double amount);
    virtual std::string toString() const;
    virtual std::string toString(char delim) const;
};

class InsecureAccount : public Account {
// ...
public:
    virtual void deposit(double amount, Date date);
    virtual std::string toString() const;
};
```

- ve výše uvedeném příkladu:
 - metoda toString ve třídě Account je *přetížena*
 - metoda toString() ve třídě InsecureAccount *předefinová*va metodu toString() ve třídě Account a zároveň *skrývá* metodu toString(char) ze třídy Account
 - metoda deposit(double, Date) ve třídě InsecureAccount *skrývá* metodu deposit(double) ze třídy Account

```
InsecureAccount iacc;
Account & refAcc = iacc;

iacc.deposit(3.50);    // chyba!
refAcc.deposit(3.50); // OK, volá se Account::deposit(double)

string s = refAcc.toString();
// OK, volá se InsecureAccount::toString()
s = iacc.toString(','); // chyba!
```

- jak zabránit skrytí metody toString(char)?

```
class InsecureAccount : Account {
// ...
public:
    using Account::toString;
// ...
};
```

Specifikátory final a override

- zavedeny v C++11
- **override** umožňuje explicitně sdělit, že zamýšlíme metodu předefinovat
 - překladač pohlídá, že typ metody sedí a že je virtuální

```
class Animal {
public:
    virtual void makeSound() const {
        std::cout << "<generic animal sound>\n";
    }
};
```

```
class Dog : public Animal {
public:
    void makeSound() override { // chyba při překladu
        std::cout << "Whoof!\n";
    }
};
```

- **final** umožňuje explicitně sdělit, že daná virtuální metoda už nesmí být v potomcích předefinována

```
class A {
public:
    virtual void f() final;
};
```

```
class B : public A {
    void f(); // chyba při překladu
};
```

- **final** je možno rovněž použít u třídy; znamená pak, že od dané třídy se nesmí dědit

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



```
class B final : public A { /* ... */ };

class C : public B { /* ... */ }; // chyba při překladu
```

Doporučení pro virtuální metody:

- rozmyslete si, zda má být daná metoda virtuální
 - bude se od dané třídy dědit?
 - je metoda určena k tomu, aby ji potomci předefinovali?
- pokud ano, pak:
 - v předkovi pište virtual
 - v potomcích můžete a nemusíte psát virtual, je ale vhodné psát override (ušetří vám to spoustu problémů)

Abstraktní třídy, rozhraní

- abstraktní třída je třída, která má alespoň jednu čistě virtuální metodu
 - čistě virtuální metody se zapisují speciální syntaxí = 0

```
class Abstract {
public:
    virtual void method() = 0;
    // ...
    virtual ~Abstract() {}
};
```

- nelze vytvářet instance abstraktní třídy
- čistě abstraktní třída: všechny metody jsou čistě virtuální (s případnou výjimkou destruktoru)
- čistě abstraktní třídě bez atributů se také říká rozhraní (interface)
 - deklaruje pouze metody, které bude možno volat
 - nenutí svým potomkům nic jiného

Vícenásobná dědičnost

- kontroverzní, výhody i nevýhody
- použití v případech, kdy jeden objekt má více charakteristik
 - časté použití spolu s rozhraními (interfaces)
 - př. kopírka může být používána zároveň jako tiskárna i skener

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

- pokud se jednotliví předkové funkčně nepřekrývají, nemusí dojít k problémům

```

class Animal { /* ... */ };
class FlyingAnimal : public Animal { /* ... */ };
class Mammal      : public Animal { /* ... */ };

class Bat : public Mammal, public FlyingAnimal { /* ... */ };

```

- co když FlyingAnimal i Mammal předefinovávají nějakou metodu z třídy Animal a Bat ji nepředefinuje?
- co když Animal obsahuje nějaké atributy?

Problém diamantu

- situace, kdy B a C dědí od A, D dědí od B a C
- D v sobě obsahuje A dvakrát

```

      D
     / \
    B   C
    |   |
    A   A

```

- objekty potomka v sobě ve skutečnosti obsahují předka, nějak takto:

```

objekt A: [atributy A]
objekt B: [atributy A | atributy B]
objekt C: [atributy A | atributy C]
objekt D: [atr. A | atr. B | atr. A | atr. C | atr. D]
           ~~~~~          ~~~~~
           zděděno od B    zděděno od C

```

```

class Animal {
    int weight;
public:
    int getWeight() const {
        return weight;
    }
};

```

// ... FlyingAnimal, Mammal, Bat jako výše ...

```
Bat theBat;
```

```
theBat.getWeight(); // CHYBA! není jasné, která getWeight() se má volat
```

```
Animal * animal = &theBat; // CHYBA! není jasné, jak přetypovat
```

- řešení: virtuální dědičnost

```
class FlyingAnimal : virtual public Animal { /* ... */ };
class Mammal       : virtual public Animal { /* ... */ };
```

- pokud B i C dědí od A virtuálně, pak neobsahují A, ale speciální ukazatel na atributy A

```

objekt A: [atributy A]
objekt B: [atributy B | ukazatel na A | atributy A]
              \-----^

```

```
objekt C: [atributy C | ukazatel na A | atributy A]
              \-----^
```

objekt D: [atr. B | atr. C | atr. D | uk. na A | atr. A]

- použitím virtuální dědičnosti už má hierarchie tvar diamantu

$$\begin{array}{cc} & A \\ & / \quad \backslash \\ B & & C \\ & \backslash \quad / \\ & D \end{array}$$

- nevýhody virtuální dědičnosti
 - všichni potomci A musí A inicializovat
 - přetypování je složitější a pomalejší
- to, že umíme problém diamantu řešit pomocí virtuální dědičnosti ještě neznamená, že bych ji měli všude používat; často je mnohem vhodnější řešit problém změnou hierarchie nebo použitím kompozice místo dědičnosti

Kompozice

- alternativa k (vícenásobné) dědičnosti
- dědičnost je vztah IS-A (Student is a Person.)
 - potomek má všechny vnější vlastnosti předka
 - potomka je možno přetypovat na předka (na studenta je možno se dívat jako na osobu)
- kompozice je vztah HAS-A (Laptop has a CPU.)
 - třída může mít jako atribut další třídu (hodnotou, referencí, ukazatelem)
 - třída může mít i víc tříd jako atributy
 - třída tím obsahuje všechny vlastnosti těchto tříd
 - ale není jejich potomkem, nemůže je zastoupit (notebook obsahuje CPU a RAM, ale není možno se na něj dívat jako na CPU nebo jako na RAM)

```

// dědičnost, v tomto případě spíš nevhodná
class Laptop : public CPU, public RAM {
// ...
};

// kompozice
class Laptop {
    CPU m_cpu;
    RAM m_ram;
public:
    Laptop(unsigned cpuFreq, unsigned ramSize) :
        m_cpu(cpuFreq), m_ram(ramSize) {}
    unsigned getCPUFreq() const {
        return m_cpu.getCPUFreq();
    }
    unsigned getRAMSize() const {
        return m_ram.getRAMSize();
    }
    // ...
}

```

- vhodnost použití kompozice a dědičnosti
 - obecně spíše preferovat kompozici
 - dědičnost tam, kde to dává smysl
 - kompozice může být kódově rozsáhlejší
 - možná i kombinace obou přístupů