

PB161 Programování v jazyce C++

Přednáška 3

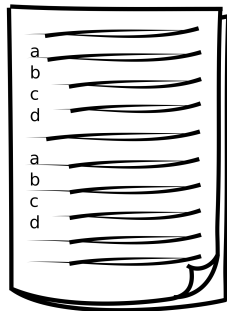
Reference, const
Přetěžování funkcí
Statické atributy a metody
Dědičnost a kompozice

Nikola Beneš

5. října 2015

Vnitrosemestrální test

- v době přednášky 2. 11. 2015
- dvě skupiny (12–13, 13–14)
- papírový odpovědník, max. 20 bodů
- náplň předchozích přednášek



Reference, const

```
void swap(int * x, int * y) {  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

```
swap(&a, &b);
```

Problémy s ukazateli

- neinicializovaný ukazatel
- NULL (v C++11 `nullptr`)

Odbočka: Proč `nullptr`?

Syntax: *typ & proměnná*

- alternativní jméno pro objekt/proměnnou (alias)

```
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
```

Reference vs. ukazatele

Reference musí být inicializovaná

```
int * ptr; // OK
```

```
int & ref; // CHYBA!
```

Reference vs. ukazatele

Reference musí být inicializovaná

```
int * ptr; // OK  
int & ref; // CHYBA!
```

Reference vždy odkazuje na skutečný objekt

```
int * ptr = nullptr; // s referencemi nic takového nejde
```

Reference vs. ukazatele

Reference musí být inicializovaná

```
int * ptr; // OK
int & ref; // CHYBA!
```

Reference vždy odkazuje na skutečný objekt

```
int * ptr = nullptr; // s referencemi nic takového nejde
```

Reference se nedá „přesměrovat“

```
int a = 1, 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 vloží hodnota 2
ref = &b;     // CHYBA!
```


Reference vs. ukazatele (pokr.)

- není možno se přímo odkazovat na proměnnou typu reference
- reference 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
```

Bezpečnější alternativa ukazatelů

Předávání parametrů referencí

- umožňuje měnit hodnotu proměnné

```
void swap(int & x, int & y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

```
int a = 3;  
int b = 7;
```

```
swap(a, b);
```

Konstantní reference

- klíčové slovo `const`

```
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 větších objektů

- deklarujeme záměr neměnit proměnnou (read-only)

```
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;  
}
```

Konstantní reference: použití (pokr.)

Předávání konstantní referencí vs. hodnotou

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

```
BigObject y;
```

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

Poznámka: situace je trochu složitější (optimalizace v překladačích, C++11 a *move semantics*, apod.)

Reference a (l/p)-hodnoty

l-hodnoty vs. p-hodnoty

- l-hodnota (l-value) je něco, co může stát na levé straně přiřazení
 - má to adresu
- p-hodnota (r-value) je něco, co může stát na pravé straně přiřazení
 - např. číselná hodnota, dočasný objekt
- konstantní reference „chytá“ všechno, nekonstantní jen l-hodnoty

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

```
int a = 7;  
lfun(a);    // OK  
lfun(17);   // CHYBA!
```

```
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

Jak inicializovat?

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


Reference jako atributy

Jak inicializovat?

```
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) {}  
};
```

Konstatní metody

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)

Konstantní metody a reference

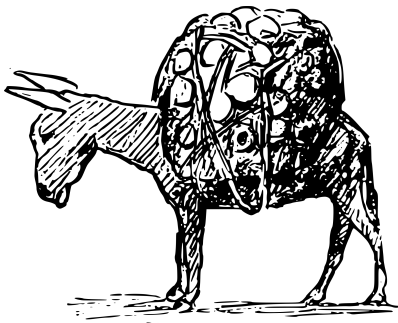
```
void funR(const Person & person) {  
    cout << person.getAge(); // OK  
    person.setAge(18);       // CHYBA!  
}
```

- stejně to funguje s ukazateli

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

- doporučení: označovat jako **const** všechny metody, které nehodlají měnit vnitřní stav objektu

Přetěžování funkcí



Přetěžování (overloading)

- různé funkce/metody se stejným jménem, ale různou implementací
- musí se lišit typem nebo počtem parametrů nebo `const`

```
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! V C++ nestačí, pokud se liší pouze návratovým typem; proč?

Přetěžování (overloading) – pokr.

Pravidla pro přetěžování

- přesná shoda
- rozšíření typu
- typové konverze, ...

```
class File {  
    // ...  
public:  
    void write(int num);  
    void write(std::string str);  
    // ...  
};  
  
File f;  
    // ...  
f.write("Hello"); // volá File::write(std::string)  
f.write(0.0);     // volá File::write(int);
```

Přetěžování a reference

- nekonstantní reference 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
```

- přetěžování pro referenci a hodnotu – nejednoznačnost (ambiguity)

Přetěžování a NULL

- v C++03: NULL je definováno jako 0

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

`f(NULL);` *// zavolá se PRVNÍ funkce!*

- 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`

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):

```
void fun(int, char) → _Z3funic
```

```
int fun(int &) → _Z3funRi
```

Implicitní parametry

Parametry s implicitní hodnotou

- funkce, metody, konstruktory, ...
- musí být nejvíce vpravo v seznamu parametrů

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

```
f(3, 4, 5); // zavolá se 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!
```

Implicitní parametry (pokr.)

Oddělení deklarace a definice

```
// soubor.h
```

```
void f(int = 10, int = 20);
```

```
// soubor.cpp
```

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

Statické atributy a metody

- položky tříd patří objektům (instancím třídy)
- statické položky třídy:
 - 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;  
    }  
}
```

Statické atributy a metody (pokr.)

Inicializace statických atributů: mimo deklaraci třídy

```
int Person::count = 0;
```

```
Person franta("Franta", 20);
```

```
Person pepa("Pepa", 21);
```

```
Person jimmy("James Bond", 42);
```

Volání statické metody

- bez konkrétního objektu

```
cout << Person::getCount() << endl; // vypíše 3
```

Statické atributy a metody (pokr.)

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

Motivace: Rozhraní

Příklad: tiskárny

```
// 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();
};
```


Použití rozhraní IPrinter

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

Poznámka: o **new** a **delete** si řekneme příště

Motivace: 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 {  
    // ...  
};
```

- 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

Dědičnost a přístupová práva

```
class A {  
public: // veřejné, vidí všichni  
    int x;  
protected: // vidí potomci  
    int y;  
private: // soukromé, vidí jen instance třídy A  
    int z;  
};
```

```
class B : public A    { /* ... */ };  
class C : protected A { /* ... */ };  
class D : private A   { /* ... */ };
```

Dědičnost a reference

- víme: ukazateli na předka můžeme předat potomka
- podobně to funguje i s referencemi

```
class Animal { /* ... */ };  
class Dog : public Animal { /* ... */ };  
  
class Person {  
    // ...  
public:  
    void playWith(const Animal & animal);  
    // ...  
};  
  
Person tommy;  
Dog    lassie;  
tommy.playWith(lassie);
```

Dědičnost: vazba

Časná vazba

```
class Animal {
public:
    void makeSound() const { std::cout << "<generic 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 sound>"
ptrAnimal->makeSound();        // "<generic sound>"
```

Dědičnost: vazba (pokr.)

Pozdní vazba

```
class Animal {  
public:  
    virtual void makeSound() const { std::cout << "<generic sound>\n";  
};  
class Dog : public Animal {  
public: // zde může i nemusí být virtual  
    virtual void makeSound() const { std::cout << "Whoof!\n"; }  
};
```

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

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

Dědičnost: vazba (pokr.)

Pozor! typ předefinované metody musí být úplně stejný včetně `const`

```
class Animal {  
public:  
    virtual void makeSound() const { std::cout << "<generic sound>\n";  
};  
class Dog : public Animal {  
public: // deklarujeme NOVOU metodu, která skrývá tu starou!  
    virtual void makeSound() { std::cout << "Whoof!\n"; }  
};
```

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

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


Overloading, overriding, hiding

Přetěžování (overloading)

- deklarace několika metod se stejným názvem a různými parametry

Předefinování (overriding)

- nahrazení definice virtuální metody ve třídě předka novou definicí ve třídě potomka

Skrývání (hiding)

- v potomkovi deklarujeme metodu stejného jména jako v předkovi a buď není virtuální nebo má jiné parametry, případně se liší kvalifikátorem `const`

Overloading, overriding, hiding (pokr.)

```
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;  
};
```

Overloading, overriding, hiding (pokr.)

```
InsecureAccount iacc;  
Account & refAcc = iacc;  
  
iacc.deposit(3.50);    // chyba!  
refAcc.deposit(3.50); // OK(?), volá 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;  
    // ...  
};
```

Overriding v C++11

Specifikátor `override`

- umožňuje explicitně sdělit, že zamýšlíme metodu předefinovat

```
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";  
    }  
};
```

Overriding v C++11

Specifikátor `final`

- metoda nesmí být předefinována; od třídy se nesmí dědit

```
class A {  
public:  
    virtual void f() final;  
};  
  
class B : public A {  
public:  
    void f(); // CHYBA! při překladu  
};  
  
class D { /* ... */ };  
class E final : public D { /* ... */ };  
class F : public E { /* ... */ }; // 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

- má alespoň jednu čistě virtuální metodu (speciální syntaxe = 0)
- nelze vytvářet instance

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

Čistě abstraktní třída

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

Rozhraní (interface)

- čistě abstraktní třída bez atributů
- deklaruje pouze metody, které bude možno volat

Vícenásobná dědičnost

- kontroverzní, výhody i nevýhody
- použití v případech, kdy jeden objekt má více charakteristik
 - časté použití spolu s rozhraními (interfaces)

```
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 { /* ... */ };
```


Vícenásobná dědičnost: problém diamantu

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

// ... FlyingAnimal, Mammal, Bat jako předtím ...

```
Bat theBat;  
theBat.getWeight(); // CHYBA! není jasné, která getWeight()  
                    // se má volat  
Animal * animal = &theBat; // CHYBA! není jasné,  
                             // jak přetypovat
```

Vícenásobná dědičnost: problém diamantu (pokr.)

Řešení: virtuální dědičnost

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

Nevýhody:

- všichni potomci Animal musí Animal inicializovat
- volání metod Animal a přetypování je složitější a pomalejší

Alternativy:

- změna hierarchie
- kompozice místo dědičnosti

Dědičnost je vztah IS-A

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

Kompozice je vztah HAS-A

- 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

Kompozice (pokr.)

Příklad

```
// 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

Reference

- alternativa k ukazatelům, použití pro volání odkazem

Přetěžování funkcí a metod

- musí se lišit počtem nebo typy parametrů nebo `const`

Statické atributy a metody

- patří třídě samotné, ne instancím (ty k nim mohou přistupovat)

Dědičnost

- přístupová práva
- časná a pozdní vazba
- overloading X overriding X hiding
- abstraktní třídy a rozhraní
- vícenásobná dědičnost
- kompozice jako alternativa dědičnosti