5. Polymorfismus, dědičnost, vztahy mezi objekty

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Statické a instanční atributy

- Instanční (členské) atributy:
  - každá instance má své vlastní instanční atributy
- Statické (třídní) atributy:
  - deklarované klíčovým slovem **static**, sdíleny instancemi třídy,
  - musí mít rezervovanou pamět' – je požadována definice mimo deklaraci třídy,
  - přístup se řídí standardními pravidly zapouzdření

Příklad

```
1  class T {
2    int a; // instanční členská proměnná
3    static int cnt; // statická členská proměnná
4  public:
5    T (int x) {a = x; cnt++;}
6    ~T () {cnt--;}
7  };
9  int T::cnt = 0; // definice (rezervace paměti)
```

Statické a instanční metody

- Instanční metody:
  - mohou být volány na existující objekt – instanci
  - mají přístup k instančním proměnným a mohou volat jiné instanční metodám, a to jak jak přímo v instanci, tak přes ukazatel `this`,
  - mají přístup k třídním proměnným a metodám.
- Statické (třídní) metody:
  - nemají žádnou implicitní instanci, proto ani implicitní členské proměnné, ani ukazatel this nejsou k dispozici.
  - mají přístup k třídním proměnným a mohou volat jiné třídní metody.

Příklad

```
1  class T {
2    static int cnt;
3  public:
4    static int getCount () {return cnt;}
5  };
6  cout << T::getCount (); // volání statické metody
```

Část I

Principy objektově orientovaného programování (OOP)

Principy OOP

- Zapouzdření
- Abstrakce
- Dědičnost
- Polymorfismus

Zapouzdření (Encapsulation)

- data i kód na jednom místě
- objekt komunikuje jen skrze rozhraní
- umožňuje skrýt vnitřní reprezentaci dat

Výhody

- možnost změnit vnitřní implementaci
- ochrana proti chybám
- nutí rozvrhnout program do nezávislých částí
- umožňuje další OOP vlastnosti

Abstrakce

- souvisí se zapouzdřením
- umožňuje pracovat s ideální představou, nezatěžuje programátora implementačními detaily

Datová abstrakce

- použití dat bez znalosti skutečného umístění/reprezentace
- příklad: soubor na disku, na serveru, ...

Funkční abstrakce

- uživatel nemusí znát detaily implementace
- příklad (strategická hra): zpráva *zaútoč* pro různé druhy jednotek

I. Principy objektově orientovaného programování (OOP)

Polymorfismus

Dědičnost

Statická a dynamická vazba

Vztahy mezi objekty

Polymorfismus

- Obecný princip polymorfismu
  - možnost psát obecný kód, který funguje s různými typy objektů
- Různé realizace polymorfismu
  - přetěžování funkcí (ad-hoc polymorfismus)
  - přetěžování operátorů neboli provedení rozdílné operace v závislosti na typu operandů
  - generické programování (C++: šablony, parametrický polymorfismus)
  - dědičnost (podtypový polymorfismus, subtype polymorphism)
- Polymorfimus lze dále rozdělit na
  - **statický** – rozhodnutí o volání vhodné funkce je provedeno již při překladu
  - **dynamický** – rozhodnutí o volání vhodné funkce provedeno až za běhu programu (tj. dynamicky pomocí virtuálních funkcí, tzv. late binding)

I. Principy objektově orientovaného programování (OOP)

Polymorfismus

Dědičnost

Statická a dynamická vazba

Vztahy mezi objekty

Dědičnost v C++

Dědičnost

- způsob, jak vytvořit **IS-A** vztah mezi objekty
- potomci (odvozené třídy, podtřídy) dědí od rodiče (bázové třídy, super třídy)

V C++

- je možno dědit od více předků,
- nelze však přímo dědit dvojmo od jedné třídy,
- předek musí být plně deklarován,
- dědí se metody a datové položky,
- nedědí se konstruktory, destruktor a přetížený operátor =,
- při dědění lze měnit přístupová práva k datovým složkám i metodám.

Odvozené třídy

```
class T1 : public T {
  // deklarace nových členských proměnných
  // deklarace nových členských funkcí (metod)
  // deklarace přepsaných metod
};
```

- Členské proměnné v odvozené třídě:
  - existující proměnné jsou vždy zděděny,
  - existující proměnné nemohou být odebrány,
  - datové typy existujících proměnných nemohou být změněny,
  - mohou být přidány nové proměnné.
- Metody v odvozených třídách:
  - existující metody jsou zděděny
  - existující metody mohou být přepsány (jiná implementace)
  - mohou být přidány nové metody
  - mohou být virtuální (automaticky se volá metoda potomka)

Odvozené třídy – viditelnost

- Zachování viditelnosti:

```
class T1 : public T { ... };
```

  - členské proměnné a funkce (metody) zděděné z `T` mají stejnou viditelnost v `T1`.

- Změna viditelnosti všech zděděných členů na `private`:

```
class T1 : private T { ... };
```

  - zděděné členy nejsou mimo `T1` viditelné,
  - efektem je dosažení dědičnosti členů, ale potlačení polymorfismu.

- Když není specifikována viditelnost:

```
class T1 : T { ... };
// class T1 :  private T  ...  ;
struct T1 : T { ... };
// struct T1 :  public T  ...  ;
```

Dědičnost – příklad 1/3

- Problém – vývoj třídy reprezentující čítač (čítající celá čísla).
- Čítač má čítat s určitým modulem `m`.
- Už dříve jsme vyvinuli celočíselný čítač – třídu `Counter`.
- Nová třída jen rozšíří (vylepší) funkci této existující třídy.
- Novou třídu odvodíme z existující třídy prostřednictvím dědičnosti:
  - zdědí se všechny existující položky (členské proměnné),
  - lze přidat nové položky,
  - existující metody lze zdědit nebo je přepsat (override),
  - lze přidat nové metody.

Dědičnost – příklad 2/3

```
1  class Counter {
2    int value, load;
3  public:
4    void inc () { value++; }
5    void reset () { value = load; }
6    int get () const { return value; }
7    Counter (int val) : load (val) { reset (); }
8  };
10 class CounterMod : public Counter {
11   int mod;
12 public:
13   void inc ();
14   CounterMod (int val, int modulus);
15 };
```

Dědičnost – příklad 3/3

```
1  // volání konstruktoru předka
2  CounterMod::CounterMod (int v, int m) : Counter (v % m) { mod = m; }
4  void CounterMod::inc () {
5    Counter::inc (); // volání metody inc () předka
6    value = value % mod;
7  }
8  // ...
9  Counter a (0);
10 CounterMod b (0, 5);
11 // ...
12 for (int i = 0; i < 10; i++, a.inc(), b.inc()) {
13   std::cout << "a: " << a.get () << "\t";
14   std::cout << "b: " << b.get () << "\n";
15 }
```

lec07/02-counter.cpp

Odvozené třídy a operátor přiřazení

- Instance odvozené třídy (potomka) může být přiřazena předkovi.
- Instance bázové třídy (předka) nemůže být přiřazena potomkovi.

```
1  class T { ... };
2  class T1 : public T { ... };
3  class T2 : public T { ... };
4
5  T x; T1 x1; T2 x2;
6  T *p; T1 *p1; T2 *p2;
8  x = x1; x = x2;  // obě přiřazení ok
9  x1 = x; x1 = x2; // obě přiřazení chybně
10 p = p1; p = p2;  // obě přiřazení ok
11 p1 = p; p1 = p2; // obě přiřazení chybně
```

Konstruktory a destruktory

- konstruktory předků se volají před vstupem do těla konstruktoru potomka,
- je-li více předků, jejich konstruktory se volají v pořadí, v jakém byly napsány,
- destruktor předka je volán až po dokončení těla destruktoru potomka,
- je-li předkem odvozená třída, aplikují se tato pravidla rekurzivně.

Virtuální dědění

- pokud se shodují názvy složek od různých předků případně název složky předka a potomka, lze je odlišit `::`,
- problém nastává např. při opakovaném dědění:

```
class A {int a};
class B:A {int b};
class C:A {int c};
class D:B, C {int d};
```

- neexistenci dvou výskytů třídy A v třídě D lze zařídit pomocí **virtuálního dědění**:
  - stačí předka A definovat ve třídách B a C jako virtuálního:

```
class B: virtual A {int b};
```

  - je-li třída děděna virtuálně i nevirtuálně, pak je v potomkovi obsažena jednou za všechna virtuální dědění a jednou za každé nevirtuální,
  - v konstruktoru se volají nejprve konstruktory virtuálních předků.

I. Principy objektově orientovaného programování (OOP)

Polymorfismus

Dědičnost

Statická a dynamická vazba

Vztahy mezi objekty

Statická a dynamická vazba

```
1  Counter c (0);
2  CounterMod cm (0, 5);
3  // ..
4  c.inc ();  // Counter::inc()
5  cm.inc (); // CounterMod::inc()
6  c = cm;
7  c.inc ();
```

- Operace je určena datovým typem proměnné stojící vlevo od operátoru `.` nebo `->`, zde tedy typem proměnné `c`.
- Metoda je vybrána v době kompilace.
- Statická vazba metod.

Statická a dynamická vazba

```
1  Counter * p = new Counter (0);
2  CounterMod * pm = new CounterMod (0, 5);
3  //..
4  p->inc ();  // Counter::inc ()
5  pm->inc (); // CounterMod::inc ()
6  delete p;
7  //..
8  p = pm;
9  p->inc (); // Counter::inc ()
```

- Operace je opět určena datovým typem proměnné `p`.
- Metoda je vybrána v době kompilace.
- Opět statická vazba metod.

Statická a dynamická vazba

Statická vazba

- je rychlejší, ale neumožňuje polymorfismus
- volající musí rozlišit objekty a jejich typy,
- volající musí mít znalost všech možných odvozených tříd,
- volající (vyšší úroveň abstrakce) se musí starat o implementační detaily objektů, které používá.

Dynamická vazba

- určuje volanou metodu až v době výpočtu, na základě objektů, které jsou zpracovávány
- volání je trochu pomalejší,
- volající zpracovává objekty (např. zde čítače) a provádí nějaké operace. Nemusí rozlišovat typy objektů při provádění operací,
- existující kód není třeba modifikovat, pokud do programu přidáme nové odvozené třídy.

Statická a dynamická vazba – příklad 1/4

- Funkce `callCounter` bude měnit a vypisovat stav instance čítače
- Funkce bude pracovat s každým objektem odvozeným z třídy `Counter`
- Třída `Counter` bude používat dynamickou vazbu

```
1  void callCounter (Counter * x, int inter) {
2    for (int i = 0; i < iter; i++, x->inc()) {
3      std::cout << x->get() << "\n";
4    }
5  }
6  //..
7  callCounter (new Counter(0), 10);
8  callCounter (new CounterMod(0, 5), 10);
```

Statická a dynamická vazba – příklad 2/4

```
1  class Counter
2  {
3  protected:
4    int value;
5    int load;
6  public:
7    virtual void increment ( ) { value++; }
8    virtual void decrement ( ) { value--; }
9    void reset ( ) { value = load; }
10   int get() const { return value; }
11   Counter(int val):load (val) { reset (); }
12 };
```

- Klíčové slovo `virtual` indikuje dynamickou vazbu konkrétní metody.

Statická a dynamická vazba – příklad 3/4

```
1  class CounterMod : public Counter {
2    ...
3    virtual void increment () {
4      Counter::increment ();
5      value %= mod;
6    }
7    virtual void decrement () {
8      Counter::decrement ();
9      value %= mod;
10   }
11 };
```

- Klíčové slovo `virtual` zde není třeba – metody jsou automaticky dynamicky vázány (zděděno po předkovi); může ale zvýšit čitelnost zdrojového kódu.

Statická a dynamická vazba – příklad 4/4

```
1  ...
2  int main()
3  {
4    Counter c (0);
5    CounterMod cm (0, 24);
6    cout << "Counter" << endl;
7    callCounter (&c);
8    cout << "CounterMod" << endl;
9    callCounter (&cm);
10   return 0;
11 }
```

Statická a dynamická vazba

- Mohlo by být rozhraní funkce `callCounter` změněno takto?

```
1 void callCounter (Counter & x)
```

  Ano, program by pracoval správně.

- A co takto?

```
1 void callCounter (Counter x)
```

  Nikoli, program bude pracovat s operacemi třídy `Counter` (jako kdyby nebyla dynamická vazba).

  Proč?

Statická a dynamická vazba

- Když je objekt předán přes ukazatel (nebo referenci), kód ve funkci `callCounter` pracuje s originálním objektem:
  - má originální instanci,
  - má originální interface.
- Když je objekt předán jako kopie objektu, pak:
  - je vytvořena instance `Counter` (kopírující konstruktor),
  - nová instance je objekt typu `Counter`,
  - ten má interface `Counter` a metody `Counter`.
- To není překvapující: objekt se chová jako `Counter`, protože to je instance `Counter`.
- Proč se v tomto chování C++ liší od Javy (a dalších jazyků)?
  - Java nemá možnost předávat objekty hodnotou.
  - Java vždy předává pouze referencí.

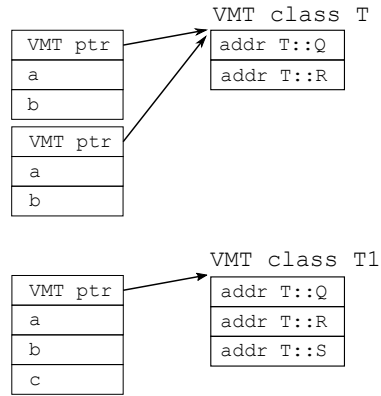
Jak dynamická vazba pracuje

- Při dynamické vazbě je metoda určena v době běhu programu:
  - objekty musí "znát" svoji třídu, aby bylo možno metodu této třídy najít,
  - metoda musí být vybrána velmi rychle.
- C++ používá VMT (Virtual Method Table) k určení metody v konstantním čase:
  - VMT je připravena pro každou třídu s virtuálními funkcemi; tabulka je generována kompilátorem,
  - VMT obsahuje pole adres metod s dynamickou vazbou,
  - každý objekt s dynamickou vazbou metod má ukazatel na svou VMT, tento ukazatel je inicializován konstruktorem,
  - metody jsou ve VMT uspořádány tak, že jméno metody koresponduje s offsetem v tabulce,
  - pro odvozené třídy se zachovává pořadí metod v tabulce.
- Volání virtuální metody znamená index ve VMT (2x dereference) a volání kódu referencovaného tímto ukazatelem.

Jak dynamická vazba pracuje

```
class T {
  int a, b;
public:
  void P ();
  virtual void Q ();
  virtual void R ();
};

class T1 : public T {
  int c;
public:
  virtual void R ();
  virtual void S ();
  void U();
};
```



Jak dynamická vazba pracuje

- Ve VMT jsou umístěny jen virtuální metody.
- Metoda `T::Q` není přepsána, její adresa je převzata do VMT třídy `T1`.
- Metoda `Q` má přiřazen index `0` v VMT, metoda `R` má index `1`.
- Tyto indexy musí být zachovány i v odvozených třídách (např. v `T1`).
- Nové metody (např. `S`) jsou přidány na konec. Metoda `S` bude mít index `2`, což bude zachováno i ve všech potomcích třídy `T1`.
- Struktura VMT je jednoduchá díky jednoduchému dědění (jeden předek). V případě vícenásobného dědění (C++ vícenásobné dědění umožňuje) se struktura VMT komplikuje.

Abstraktní třídy

- slouží pro implementaci obecného předka dalších tříd,
- některé metody takovéhoto předka nemá smysl definovat:
  - nazývá se čistě virtuální, místo těla má řetězec =0
- nesmí být vytvořena instance, pointer nebo reference však ano.

Příklad

```
1  struct Base  {
2    virtual void show() = 0;
3  };
5  struct Derived: public Base {
6    void show() { std::cout << "derived\n"; }
7  };
8  //..
9  Base *bp = new Derived();
10 bp->show();
```

I. Principy objektově orientovaného programování (OOP)

Polymorfismus

Dědičnost

Statická a dynamická vazba

Vztahy mezi objekty

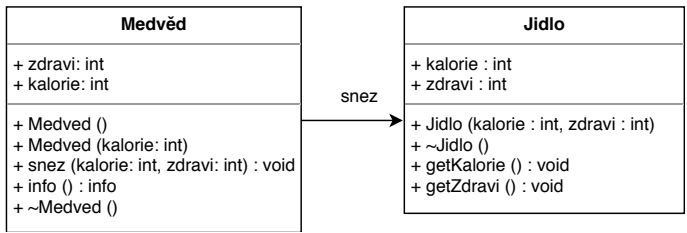
Vztahy mezi objekty

- Objekty mohou obsahovat jiné objekty
  - agregace
  - kompozice
- Definice třídy může být založena na již existujícím předpisu
  - Bázová třída (super class) a odvozené třídy
  - Vztah je přenesen do instancí
- Objekty spolu mohou komunikovat pomocí metod, které jsou jim přístupné

Asociace

- Asociace je nejvolnějším typem vazby.
  - objekty existují nezávisle na sobě
  - objekty se propojují jen na krátkou dobu (obvykle po dobu zavolání metody)
- Je využívána zejména pro předání hodnot mezi jednotlivými objekty

Příklad



lec07/03-medved-jidlo.cpp

Implementace vazeb – agregace

- Agregace (zahrnutí, obsažení) popisuje dlouhodobý vztah objektů
- Příkladem agregace může být vztah `Učitel` – `Žák`
  - Každý žák musí mít přiřazeného učitele. Ten bude právě jeden.
  - Učitel může učit jednoho a více studentů.
  - Žák může mít souseda, a to nejvýše jednoho.
  - Žák nemusí mít souseda.
  - Tím sousedem je opět nějaký žák.
- Zřejmě
  1. Učitel i žáci existují nezávisle na sobě. Není zde rozhodně žádná závislost ve smyslu: když vzikne nový žák, vznikne i jeho spolužák, případně učitel. Ani opačně: když zanikne učitel (odejde), nezaniknou žáci, atp. V tomto ohledu není nutné nic řešit.
  2. Vazby jsou delší než zavolání jedné metody. Spolužáci jsou spolužáky do té doby, než je jeden z nich zrušen nebo se přesadí. Definované vazby je tedy nutné dlouhodobě ukládat.

`lec07/04-ucitel-zak.cpp`

Implementace vazeb – kompozice

- Umožňuje sestavit komplexní objekt z několika dílčích objektů
- Objekty jsou spolu pevně svázány a nemohou bez sebe existovat
- Příkladem agregace může být vztah `Auto` – `Motor`
  - Konstruktor auta zajistí vytvoření motoru a destruktor jeho zrušení
  - Tak bude splněna podmínka, aby motor byl nedílnou součástí auta.

```
class Motor {
  // ..
};
class Auto {
  Motor *m;
public:
  Auto () { m = new Motor (); }
};
```

lec07/05-auto-motor.cpp