2. Třídy a objekty

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Třídy a objekty
  - Třídy
  - Objekty
  - Polymorfismus
  - Dědičnost
  - Statická a dynamická vazba
  - Vztahy mezi objekty
- Část 2 – Zadání 1. domácího úkolu

Část I

Třídy a objekty

I. Třídy a objekty

Co je to třída?

- Třída je abstrakcí entity reálného světa.
- Příklad: třída Automobil:
  - všechny automobily mají nějaké společné vlastnosti
    - obsah motoru,
    - barvu,
    - ...
  - všechny automobily mají nějaké společné rozhraní (interface)
    - lze je nastartovat,
    - lze je rozjet nebo zastavit,
    - ...
- Objekt je tzv. instance třídy.
- Příklady instance třídy Automobil:
  - Škoda Octavia,
  - WV Passat,
  - ...

Pohled programátora

- Třída je popisem datového typu:
  - jméno,
  - data – členské proměnné (položky, atributy),
  - interface – členské funkce (metody).
- Třídy jsou vyvíjeny programátory a jsou kompilovány do spustitelného programu.
- V C++ nelze vytvářet nové třídy za běhu.
- Objekty (instance tříd) jsou proměnné:
  - každý objekt má třídu,
  - v průběhu běhu programu jsou objekty vytvářeny a rušeny,
  - obvykle vytváříme více objektů/instancí stejné třídy,
  - stejně jako jiné datové typy, C++ povoluje staticky alokované objekty, dynamicky alokované objekty, pole objektů, ...

Deklarace třídy

```
class T
{
  typ a;           // atribut (členská proměnná)
  typ f ( ... ); // metoda (členská funkce)
  T ( ... );       // konstruktor
  ~T (void);       // destruktor
};
```

- Definice konstruktoru:

  `T::T ( ... ) { ... }`

- Definice destruktoru

  `T::~T ( void ) { ... }`

- Definice metody

  `typ T::f ( ... ) { ... }`

Inline metody

```
class T
{
  typ f ( ... ) { ... }
  T ( ... ) { ... }
  ~T (void) { ... }
};
```

- Inline metody jsou podobné inline funkcím

  možná rychlejší, s vyššími paměťovými nároky

- Kompilátor se může rozhodnout nezkompilovat metodu jako inline

  vypnuté optimalizace, příliš dlouhý kód, ...

- Inline metody snižují srozumitelnost deklarací
- Všimněte si, že
  - deklarace metod jsou zakončeny středníkem,
  - definice metod středníkem zakončeny nejsou.

Zapouzdření

- Pro úplnou ochranu atributů objektu je potřeba zabránit jejich modfikaci jinak než přes příslušné metody.
- Atributy a metody třídy jsou vždy přístupné z metod definovaných v této třídě, z jiných metod a funkcí však již přístupné být nemusí.
- Zapouzdření je zároveň prostředek, jak vytvořit a udržovat kontrolovatelné a vysokoúrovňové veřejné rozhraní třídy.
- Zapouzdření umožňuje:
  - nezávisle upravovat implementaci uvnitř třídy (např. zvolit efektivnější algoritmus, jinou reprezentaci dat, ...),
  - kompletně nahradit třídu bez rozbití zbytku programu (pokud nová třída dodrží veřejné rozhraní),
  - opravit chyby uvnitř třídy bez rozbití zbytku programu,
  - pracovat na vývoji tříd nezávisle (např. různými programátory zároveň).

Řízení přístupu

- Přístup k atributům a metodám je řízen pomocí **modifikátorů viditelnosti**:
  - `public` – jsou přístupné komukoli,
  - `protected` – jsou přístupné v třídě samé a v jejích podtřídách,
  - `private` – jsou přístupná jen v třídě samé.

```
class T
{
  // metody/atributy s implicitním přístupem
  // (zde private)
public:
  // metody/atributy přístupné komukoli
private:
  // metody/atributy přístupné jen ve třídě
};
```

Klíčová slova `class` a `struct`

- Obě klíčová slova mohou být použita k deklaraci třídy
- Jediný rozdíl je v implicitní viditelnosti:
  - class: `private`
  - struct: `public`

```
class T {
  // implicitni private
    int a;
  public:
    void f ();
};
```

```
struct T {
  private:
    int a;
  public:
    void f ();
};
```

```
class T {
  public:
    void f ();
  private:
    int a;
};
```

```
struct T {
  // implicitni public
    void f ();
  private:
    int a;
};
```

I. Třídy a objekty

Vytváření objektů

- Objekt je proměnná (instance třídy)
- Při vytvoření objektu je zavolán konstruktor
- Konstruktory mohou být přetíženy
- Konstruktor nelze volat explicitně na existující objekt
- Destruktor je volán automaticky, když je objekt rušen

Příklad

```
int main() {
  T x (a, b); // konstruktor s parametry (a, b)
  T y;        // konstruktor bez parametrů
  // ..
}
```

Přístup k atributům a metodám

- Metody jsou volány prostřednictvím tečkové notace
- Metody mohou být přetíženy. Platí pravidla pro přetěžování funkcí (nejlepší shoda skutečných a formálních parametrů)
- Přístup k atributům je tečkovou notací (jako struct).

Příklad

```
class T {
  public:
    void foo ( ) { ... } // metoda
    int bar;             // atribut
};
// ..
T x;
x.foo ();
x.bar = 10;
```

Dynamická alokace

- Objekty mohou být alokovány dynamicky – užitím operátoru `new`

  Operátor volá odpovídající konstruktor

- Dynamicky alokované objekty je rušeny použitím `delete`

  Operátor volá destruktor

- C alokace (malloc/free) nemůže být použita

  Tyto funkce by nezavolaly konstruktor a destruktor

Příklad

```
struct T
{
  int a;
  T (int x) {a = x;}
  ~T ();
  void f () {cout << a;}
};
```

```
int main()
{
  T *p = new T(20);
  p->f();
  p->a = 10;
  p->f();
  delete p;
}
```

Klíčové slovo `this`

- Lokální deklarace mohou být v konfliktu se jmény položek.
- Přístup k položkám lze zařídit pomocí klíčového slova `this` nebo plně kvalikovaným jménem.
- Lepší je vyhnout se konfliktu jmen (např. prefixem).

Příklad

```
struct T {
  int a;
  void f (int a);
};
void T::f (int a) { // konflikt se jménem atributu a
  T::a = a; // T::a - plně kvalifikovaný atribut
            // a - jméno parametru metody
  this->a = 10; // this - ukazatel na instanci typu T*
}
```

Konstantní objekty a konstantní metody

- Objekt nebo metoda může být označen jako konstantní (const)
  - položky konstantného objektu nemohou být modifikovány
  - konstantní metody nesmí modifikovat objekt (kontrola)
  - konstantní metody mohou být volány na jakémkoli objektu
  - nekonstantní metody nelze volat na konstantním objektu

Příklad

```
struct T {
  int a;
  T (int x) {a = x;}
  void print () {cout << a;}
  int get () const {return a;} // const metoda
};
// ..
const T x (1); // const objekt
cout << x.get(); // ok - const metoda na const objektu
x.print ();      // chyba - non-const metoda
```

Volání konstruktoru

- Konstruktor je automaticky volán při vytváření objektu
- Po vytvoření objektu už další volání konstruktoru není dovoleno
- Explicitní volání konstruktoru vytvoří nový nepojmenovaný dočasný objekt – může být použit pro výpočet (např. předání parametrů, lokální kopii, ...) a je pak zrušen.

Příklad

```
struct T {
  int p, q;
  T () {p=0; q=0;}
  T (int fp) {p=fp; q=0;}
};
// ..
T a, b (1);
a.T (1); // chyba, dalsi volani konstruktoru
a = T (20); // je vytvoren novy docasny objekt, ten je
    zkopirovan do a, potom je docasny objekt zrusen
```

Implicitní konstruktor

- Konstruktor, který lze volat bez parametru, tj.:
  - nemá žádný formální parametr, nebo
  - jeho všechny formální parametry mají implicitní hodnoty.
- Implicitní konstruktor se použije při vytvoření nového objektu, nejsou-li uvedeny parametry.
- Když ve třídě není žádný konstruktor, systém automaticky generuje implicitně (prázdný) konstruktor.

Příklad

```
struct T {
  T (int a = 0) { ... } // implicitni konstruktor
};
// ..
T a;
T b[10]; // implicitni konstruktor volan 10x
T *p = new T;
```

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 paměť' – je požadována definice mimo deklaraci třídy,
  - přístup se řídí standardními pravidly zapouzdření

Příklad

```
class T {
    int a; // instanční členská proměnná
    static int cnt; // statická členská proměnná
  public:
    T (int x) {a = x; cnt++;}
    ~T () {cnt--;}
};
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

```
class T {
    static int cnt;
  public:
    static int getCount () {return cnt;}
};
cout << T::getCount (); // volání statické metody
```

I. Třídy a objekty

Polymorfismus

- Polymorfismus je vlastnost OOP programovacího jazyka která umožňuje:
  - jednomu objektu volat jednu metodu s různými parametry

    parametrický polymorfismus
  - přetěžování operátorů neboli provedení rozdílné operace v závislosti na typu operandů
  - objektům odvozeným z různých tříd volat tutéž metodu se stejným významem v kontextu jejich třídy
- 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)

Přetížené operátory

- Pro přetěžování operátorů jsou jistá omezení
  - přetíženy mohou být jen existující operátory,

    Není možné zavést nový operátor, např. operátor $.
  - aritu, asociativitu a prioritu operátorů nelze změnit,

    Např. operátor += musí být binární.
  - nelze přetěžovat operátory vestavěných typů.

    Nelze změnit způsob sčítání čísel typu integer.
- Přetížení je třeba si dobře rozmyslet
  - přetížené operátory mají význam v matematice (např. naše komplexní čísla, velká čísla, vektory, ...) a řetězce (konkatenace – spojování),
  - kolekce (jako vektory, množiny, tabulky, ...) používají přetížené operátory pro přístup k uloženým hodnotám,
  - jiné třídy mohou vyžadovat přetížené operátory `<<`, `>>` a `=`,
  - jste-li na pochybách, dejte přednost metodě před přetíženými operátory.

Přehled operátorů

- Operátory, které lze přetížit

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`
`~`	`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`
`/=`	`%=`	`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`
`--`	`->*`	`,`	`->`	`[]`	`()`	`new`	`delete`
`new []`	`delete[]`						

- Operátory, které nelze přetížit

`.`	`.*`	`::`	`?:`	`sizeof`

Binární operátory

`+, -, *, ..., +=, -=, ...`

- zápis operace: `x @ y`
- signatura: `T1 @ T2 -> T3`
- metoda v T1: `T3 operator @ (T2)`
- alt. volání: `x.operator @ (y)`
- funkce: `T3 operator @ (T1, T2)`
- alt. volání: `operator @ (x, y)`

Přetížení operátoru `@` automaticky nepřetíží operátor `@=`. Je-li to požadováno, operátor musí být přetížen separátně.

Příklad přetížení binárního operátoru

```
struct cplx {
  float im, re;
  cplx operator+(cplx & a) {
    cplx tmp;
    tmp.im = this->im + a.im;
    tmp.re = this->re + a.re;
    return tmp;
  }
};
cplx operator-(cplx & a, cplx & b) {
  cplx tmp;
  tmp.re = a.re - b.re;
  tmp.im = a.im - b.im;
  return tmp;
}
```

lec02/01-cplx-pretizeni.cpp

Unární prefixové operátory

`+, -, *, &, ~, !, ++, --`

- zápis operace: `@ x`
- signatura: `@ T1 -> T2`
- metoda v T1: `T2 operator @ ()`
- alt. volání: `x.operator @ ()`
- funkce: `T2 operator @ (T1)`
- alt. volání: `operator @ (x)`

Příklad přetížení prefixového unárního operátoru

```
struct cplx
{
  float im, re;
  // komplexně sdružené číslo
  cplx & operator~()
  {
    this->im = -1 * this->im;
    return *this;
  }
};
// test nulovosti
bool operator!(cplx & a)
{
  return a.re == 0 && a.im == 0;
}
```

lec02/XX-pretizeni-unarni.cpp

Unární postfixové operátory

`++, --`

- zápis operace: `x @`
- signatura: `T1 @ -> T2`
- metoda v T1: `T2 operator @ (int)`
- alt. volání: `x.operator @ (0)`
- funkce: `T2 operator @ (T1, int)`
- alt. volání: `operator @ (x, 0)`

Přiřazení

- zápis operace: `x = y`
- signatura: `T = T1 -> T`
- metoda v T1: `T & operator = (const T1 &)`
- alt. volání: `x.operator = (y)`
- funkce: -
- alt. volání: -

- Obvykle bývají `T1` a `T` stejné typy.
- Návratová hodnota může být buď typu `T &`, nebo `void`.

Indexování

- zápis operace: `x[y]`
- signatura: `T1[T2] -> T3`
- metoda v T1 (a): `T3 & operator [] (T2)`
- metoda v T1 (b): `const T3 & operator [] (T2) const`
- alt. volání: `x.operator [] (y)`
- funkce: -
- alt. volání: -

- První způsob přetížení je určen pro nekonstantní objekty a umožňuje použít výsledek jako l-value (tzn. na levé straně přiřazení).
- Druhý způsob přetížení je určen pro konstantní objekty, výsledek není modifikovatelný.

Dereference

- zápis operace: `x -> met`
- signatura: `T -> T1*`
- metoda v T: `T1* operator -> ()`
- alt. volání: `(x.operator -> ()) -> met`
- funkce: -
- alt. volání: -

Volání funkce

- zápis operace: `x (x1, x2, ..., xn)`
- signatura: `T (T1, T2, ..., Tn) -> Tx`
- metoda v T: `Tx operator () (T1, T2, ..., Tn)`
- alt. volání: `(x.operator () (x1, x2, ..., xn))`
- funkce: -
- alt. volání: -

I. Třídy a 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

- Odvozené třídy mají tuto syntaxi

```
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

- 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


```
class Counter {
  protected:
    int value, load;
  public:
    void inc () { value++; }
    void reset () { value = load; }
    int get () const { return value; }
    Counter (int val) : load (val) { reset (); }
};
class CounterMod : public Counter {
  protected:
    int mod;
  public:
    void inc ();
    CounterMod (int val, int modulus);
};
```

Dědičnost – příklad


```
CounterMod::CounterMod (int v, int m) :
  // volání konstruktoru předka
  Counter (v % m) { mod = m; }
void CounterMod::inc () {
  Counter::inc (); // volání metody inc () předka
  value = value % mod;
}
// ...
Counter a (0);
CounterMod b (0, 5);
for (int i = 0; i < 10; i++, a.inc(), b.inc())
{
  std::cout << "a: " << a.get () << "\t";
  std::cout << "b: " << b.get () << "\n";
}
```

lec02/02-counter.cpp

Odvozené třídy a operátor přiřazení

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

```
class T { ... };
class T1 : public T { ... };
class T2 : public T { ... };

T x; T1 x1; T2 x2;
T *p; T1 *p1; T2 *p2;
x = x1; x = x2;  // obě přiřazení ok
x1 = x; x1 = x2; // obě přiřazení chybně
p = p1; p = p2;  // obě přiřazení ok
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. Třídy a objekty

Statická a dynamická vazba

```
Counter c (0);
CounterMod cm (0, 5);
// ..
c.inc ();  // Counter::inc()
cm.inc (); // CounterMod::inc()
c = cm;
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

```
Counter * p = new Counter (0);
CounterMod * pm = new CounterMod (0, 5);
//..
p->inc ();  // Counter::inc ()
pm->inc (); // CounterMod::inc ()
delete p;
//..
p = pm;
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

- 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

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

Statická a dynamická vazba

```
class Counter
{
protected:
  int value;
  int load;
public:
  virtual void increment ( ) { value++; }
  virtual void decrement ( ) { value--; }
  void reset ( ) { value = load; }
  int get() const { return value; }
  Counter(int val):load (val) { reset (); }
};
```

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

Statická a dynamická vazba

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

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

  Ano, program by pracoval správně.
- A co takto?

```
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.

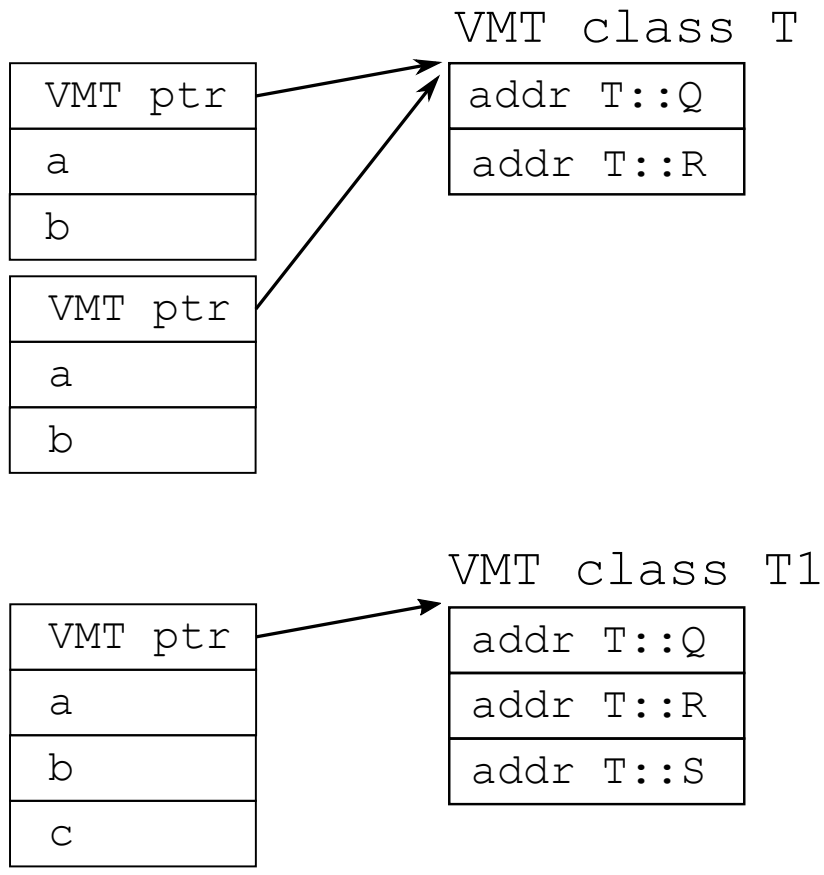
Jak dynamická vazba pracuje

- 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

```
struct Base  {
  virtual void show() = 0;
};
struct Derived: public Base {
  void show() { std::cout << "derived\n"; }
};
//..
Base *bp = new Derived();
bp->show();
```

I. Třídy a objekty

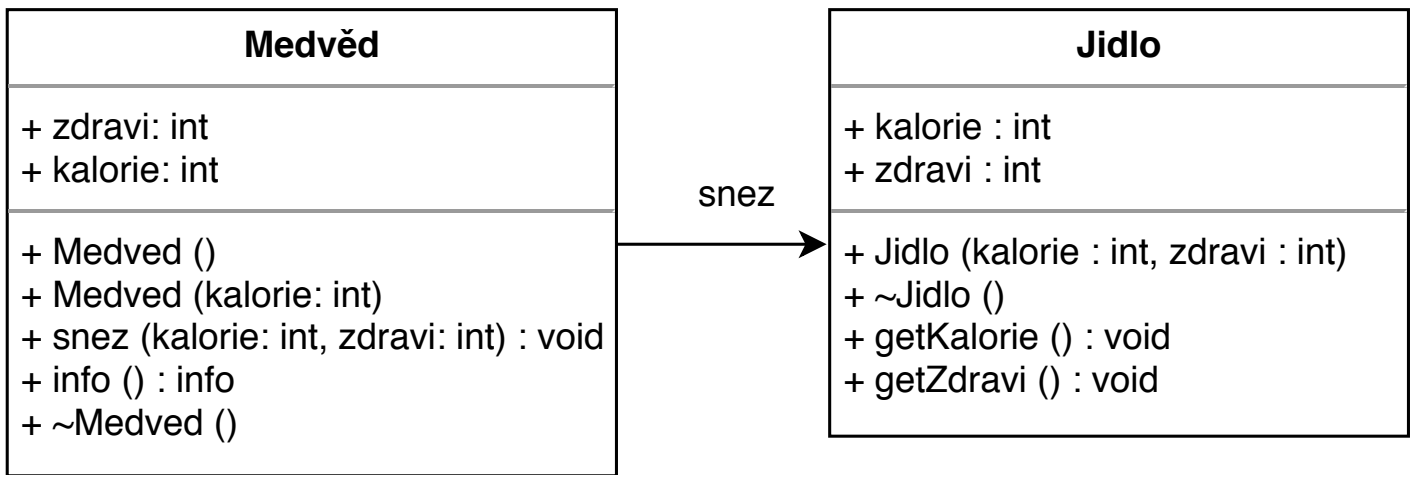
Vztahy mezi objekty

- 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



lec02/03-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.

lec02/04-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 (); }
};
```

lec02/05-auto-motor.cpp

Část II

Zadání domácího úkolu

Zadání 1. domácího úkolu (HW01)

Téma: Standardní vstup a výstup v C++

- **Motivace:** Formátování načtených dat na základě předpisu
- **Cíl:** Procvičení proudového výstupu v C++
- **Zadání:** `https://cw.fel.cvut.cz/wiki/courses/b2b99ppc/hw/01`
  - Zpracování formátovaného textu
  - Výpis ve formátu daném předpisem, Excel like tabulka
  - Výpočet funkce v tabulce
- **Termín odevzdání:** **14.3.2020, 23:59:59**