Programování v C++ 1, 6. cvičení

dědičnost, polymorfismus

Vladimír Jarý[1]

[1]Fakulta jaderná a fyzikálně inženýrská
České vysoké učení technické v Praze

Zimní semestr 2020/2021

Přehled

Shrnutí minule procvičené látky

- princip zapouzdření
  - přístupová práva
  - spřátelené funkce
- statické metody a atributy
- konstruktory
  - význam
  - inicializační část
  - kopírovací konstruktor
- nevirtuální dědění
  - přístup ke zděděným složkám
  - dědění a skládání
  - potomek může zastoupit předka
- třída `string`

Dědičnost

- v C++ vícenásobná dědičnost (potomek má více předků)
- nevirtuální dědění: předci neobsahují specifikace virtual
  - `class` *identifikátor* : *seznam_předků*{*tělo_třídy*};
  - seznam_předků:
    - identifikátor_přístupu$_{nep}$ identifikátor
    - identifikátor_přístupu$_{nep}$ identifikátor, seznam_předků
  - identifikátor přístupu: `public`, `protected`, `private`
    - `private`: zděděné složky budou v potomkovi soukromé
    - `protected`: zděděné složky budou v potomkovi chráněné
    - `public`: zděděné veřejné a chráněné složky budou v potomkovi veřejné a chráněné
    - ve všech případech budou soukromé složky předka v potomkovi nedostupné
- potomek při nevirtuálním dědění zdědí:
  - všechny atributy svých předků
  - všechny metody s výjimkou konstruktorů, destruktorů (a přetíženého operátoru =)

Poznámka: dědění a skládání

Odvodit úsečku od bodu zděděním?

- ne – úsečka *není* speciální případ bodu
- úsečka *má* dva body
- relace *is a*
- relace *has a*

Příklad:

```
1 class TUsecka{
2   public:
3     //seznam verejnych polozek
4   private:
5     A, B: TBod;
6 };
```

Příklad: studenti a zaměstnanci

```
1  class Osoba{
2  public:
3    Osoba(std::string, std::string, int);
4    void tisk();
5  protected:
6    std::string jmeno, prijmeni; int id;
7  };
8  class Student : public Osoba{
9  public:
10   Student(std::string, std::string, int, int);
11   int pocetKreditu() const{return kredity;}
12 protected:
13   int kredity;
14 };
```

Implementace metod

```
1  Osoba::Osoba(string jm, string prij, int i)
2   : jmeno(jm), prijmeni(prij), id(i)
3  {}
4
5  void Osoba::tisk(){
6    cout << jmeno << " " << prijmeni << endl;
7  }
8
9  Student::Student(string jm, string prij,
10                                int i, int kr)
11 : Osoba(jm, prij, i), kredity(kr)
12 {}
```

Třída `std::string`

- hlavičkový soubor `string`
- pro práci s posloupností znaků
- přístup ke znaku pomocí operátoru `[]`
- některé metody:
  - `length`: vrátí délku řetězce
  - `empty`: testuje, zda je řetězec prázdný
  - `substr`: vytvoří podřetězec
  - `append`: připojí řetězec na konec řetězce
    - lze použít přetížený operátor `+=`
  - `compare`: porovná řetězce
  - `find`: vyhledává v podřetězci
  - `c_str`: vrátí data jako `const char*`
- lze použít s proudy `std::cin` a `std::cout`

Základní pravidlo dědičnosti

Potomek může zastoupit předka

```
1  void posliZpravu(Osoba o, string zprava){
2   cout << "Posilam zpravu " << zprava << endl;
3   cout << "Prijemce: "; o.tisk();
4  }
5  int main(){
6   string jmeno, prijmeni;
7   cout << "Zadejte jmeno a prijmeni: " << endl;
8   cin >> jmeno >> prijmeni;
9   Student s(jmeno, prijmeni, 42, 10);
10  posliZpravu(s, "Test");
11  return 0;
12 }
```

Metody tříd `Osoba`, `Student`

```
1  Osoba::Osoba(string jm, string prij, int i)
2   : jmeno(jm), prijmeni(prij), id(i){}
3
4  void Osoba::tisk(){
5    cout << jmeno << " " << prijmeni << endl;
6  }
7
8  Student::Student(string jm, string prij,
9                                int i, int kr)
10 : Osoba(jm, prij, i), kredity(kr){}
11
12 void Student::tisk(){
13   cout << jmeno << " " << prijmeni << endl;
14   cout << "Pocet kreditu: " << endl;
15 }
```

Potomek může zastoupit předka

- jedna ze základních vlastností OOP
- potomek představuje specializaci bázové třídy
- překladač konvertuje instanci potomka na typ předka:
  1. předek je na místě přetypování dostupný
  2. přetypování je jednoznačné
- třída `Student` je potomek třídy `Osoba`, proto lze psát
  - `Student student;`
  - `(Osoba)student;`
  - `static_cast<Osoba> student;`
- stejná pravidla platí i při přetypování ukazatele na potomka na ukazatele na předka (při přetypování se ukazatel na potomka změní na ukazatel na zděděný podobjekt)
- opačné přetypování (z předka na potomka) nemusí mít vždy smysl, proto nutno explicitně předepsat

Práce s objekty prostřednictvím ukazatelů

```
1  int main(){
2      Osoba *uo;
3      uo = new Student("Tomas", "Marny", 100);
4      uo->tisk();
5      delete uo;
6      return 0;
7  }
```

- očekáváme, že se zavolá metoda `tisk` třídy `Student`
- ve skutečnosti se zavolá metoda `tisk` třídy `Osoba`

Časná a pozdní vazba

- překladač použil metodu ze statického typu (tj. `Osoba *`) ⇒ *časná vazba*
- potřeba, aby se použil dynamický typ (zde `Student *`) ⇒ *pozdní vazba*
- určování skutečného typu za běhu je náročné, proto se implicitně používá časná vazba
- pro použití pozdní vazby musíme pro metody použít modifikátor **`virtual`**
  - konstuktory a statické metody nesmí být virtuální
  - destruktory mohou být virtuální
  - metoda deklarovaná jako virtuální v předkovi je virtuální i v potomkovi
  - virtuální metody musí mít v předkovi i potomkovi stejný počet a typ parametrů a musí být stejného typu

Čistě virtuální metody, abstraktní třídy

- často se stane, že v bázové třídě nevíme, jak implementovat nějakou metodu (příklad s grafickými objekty ve skriptech: jak nakreslit obecný grafický objekt)
- metoda však v předkovi musí být uvedena (jinak by ji nešlo volat pro potomka, se kterým pracujeme prostřednictvím ukazatele na předka)
- řešení je tyto metody označit jako čistě virtuální:
  - `virtual prototyp_metody = 0;`
- implementace čistě virtuální metody až v potomkovi
- abstraktní třída: třída obsahující čistě virtuální metodu
  - nelze vytvářet instance abstraktních tříd
  - lze ale deklarovat ukazatele na abstraktní třídy
  - také lze použít jako typ parametru předávaného odkazem

Třída Go

- společný předek
- abstraktní třída
- obsahuje čistě virtuální metodu `nakresli()`
  - nevíme jak nakresli abstraktní grafický objekt

```
1 class Go{
2 protected:
3   int barva;
4 public:
5   Go(int b) : barva(b){}
6   int getBarva() const {return barva;}
7   void setBarva(int b){barva = b;}
8   virtual void nakresli() const = 0;
9 };
```

Třída Bod

- dědí atribut `barva`
- musí implementovat metodu `nakresli()`

```
1  class Bod : public Go{
2  protected:
3    int x, y;
4  public:
5    Bod(int x0, int y0, int b0) : Go(b0), x(x0), y(y0){}
6    int getX() const {return x;}
7    void setX(int x0){x = x0;}
8    int getY() const {return y;}
9    void setY(int y0){y = y0;}
10   void nakresli() const;
11 };
```

Třída Usecka

- také potomek třídy `Go`
- musí implementovat metodu `nakresli()`
- skládání objektů (úsečka má dva koncové body)

```
1  class Usecka : public Go{
2  protected:
3    Bod a1, a2;
4  public:
5    Usecka(Bod a, Bod b, int b0)
6      : Go(b0), a1(a), a2(b){}
7    Usecka(int x1, int y1, int x2, int y2, int b0)
8      : Go(b0), a1(x1, y1, b0), a2(x2, y2, b0){}
9    void nakresli() const;
10 };
```

Práce s grafickými objekty

- využití dědičnosti a polymorfismu
- objekty vznikají dynamicky za běhu (například tak, jak je uživatel kreslí)
- uložení ukazatele na vytvořený objekt do ukazatele na společného předka
  - (ukazatel na potomka může zastoupit ukazatel na předka)
- při kreslení se ukazatel dereferencuje a pozdní vazba zajistí, že se vykreslí správný typ objektu

```
1 void nakresliObrazek(Go *obr[], int n){
2   for(int i = 0; i < n; i++)
3     obr[i]->nakresli();
4 }
```

Závěr

Shrnutí

- potomek může zastoupit předka
- ukazatel na potomka může zastoupit ukazatel na předka
- časná a pozdní vazba
- virtuální metody
- čistě virtuální metody a abstraktní třídy
- grafické objekty
  - abstraktní třída `Go` jako společný předek
  - obsahuje čistě virtuální metodu `nakresli()`
  - odvozené třídy (`Bod`, `Usecka`) ji musí implementovat
  - použití ukazatele na společného předka `Go*`