Programovací jazyk C++: späť k triedam

Poznámky k prednáškam z predmetu Objektovo-orientované programovanie

Valentino Vranić
http://fiit.sk/~vranic/, vranic@stuba.sk
Ústav informatiky, informačných systémov a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

19. apríl 2016

Obsah

1 Triedy v jazyku C++

O jazyku C++

- Rozšírenie jazyka C (1972) – ale nie úplne kompatibilné[1]
- Bjarne Stroustrap, Bell Labs
- Pôvodne C with Classes, ale prvé komerčné vydanie už bolo ako C++ (1983)
- Zaviedol OO črty inšpirované Simulou 67
- Multiparadigmový jazyk: procedurálne programovanie, OOP a generické metaprogramovanie
- Aj to čo označujeme ako OOP možno rozdeliť na ďalšie paradigmy...
- Táto prednáška je predovšetkým o OO vlastnostiach jazyka C++

Pôvod tried v jazyku C++

- Kľúčové slovo **class**
- Triedy v C++ sa vyvinuli zo štruktúr definovaných ako typy v jazyku C

```
typedef struct {
   char[20] meno;
   char[20] priezvisko;
   int rocnik;
} Student;
...
Student x;
strcpy(x.meno,"Milan");
strcpy(x.priezvisko,"Milov");
x.rocnik = 1;
...
Student y = ("Dana", "Petrova", 2);
```

- Štruktúry môžu obsahovať len údaje, nie funkcie
- Ale máme ukazovatele na funkcie – a už sa to začína podobať na triedy

Členské premenné a funkcie

- Atribúty – členské premenné
- Metódy – členské funkcie
- Metódy sa v triedach len deklarujú – uvádza sa ich signatúra

```
class Student {
      char[20] meno;
      char[20] priezvisko;
      int rocnik;
   public:
      void zapis(int rocnik);
   ...
};
```

[1] http://en.wikipedia.org/wiki/Compatibility_of_C_and_C++

- Telo metódy sa definuje mimo triedy

```
void Student::zapis(int rocnik) {
   this->rocnik = rocnik;
}
```

Inštanciácia

- Objekt môžeme vytvoriť rovnako ako premennú primitívneho typu

```
Student s;
s.zapis(3);
```

- Objekt môžeme vytvoriť dynamicky

```
Student* p = new Student();
p->zapis(3);
```

Konštruktory

- Konštruktory v C++ sú podobné konštruktorom v Jave
- V predchádzajúcom príklade bol využitý implicitný[2] konštruktor
- Začleňme do triedy `Student` konštruktor, ktorý nastavuje ročník

```
class Student {
      char[20] meno;
      char[20] priezvisko;
      int rocnik;
   public:
      void zapis(int rocnik);
      Student(int rocnik);
   ...
};
...
Student::Student(int rocnik) {
   this->rocnik = rocnik;
}
```

- Dynamické vytváranie objektu vyzerá celkom ako v Jave

```
Student* p = new Student(3);
```

- Argumenty konštruktora pri statickom vytváraní objektov zadáme za identifiktorom objektu

```
Student s(2);
```

[2] default

Deštruktory

- C++ nemá priamo zabudovanú podporu garbage collection
- Objekty vytvorené dynamicky (operátorom **new**) musíme rušiť explicitne operátorom `delete`
- Ak objekt nealokuje dynamicky ďalšiu pamäť, stačí implicitný deštruktor

```
Student* p = new Student(3);
...
delete p;
```

- Inak je to ak dynamicky alokujeme pamäť (pomocou **new** alebo `malloc()`)
- Deštruktor nesie názov triedy, ktorému predchádza tilda

```
class Student {
        char[20] meno;
        char[20] priezvisko;
        int rocnik;
        char* poznamka;

    public:
        void zapis(int rocnik);
        Student(int rocnik);
        ~Student();
    ...
};
...
```

```
...
Student::Student(int rocnik) {
    this->rocnik = rocnik;
    poznamka = new char[200];
}
Student::~Student() {
    delete poznamka;
}
```

Organizácia zdrojového kódu

- C++ prevzal organizáciu zdrojového kódu z jazyka C
- Hlavičkové súbory obsahujú deklarácie funkcií (`.h`)
- Deklarácie sa sprístupňujú fyzickým zahrnutím hlavičkových súborov (`#include`) – pozor na cyklické zahrnutia
- Veci ohľadom tried sú organizované podobne
- Triedy sú deklarované v hlavičkových súboroch (`.h` alebo `.hpp`)
- Implementácie metód sú v bežných zdrojových súboroch (`.cc` alebo `.cpp`)

2 Základný model dedenia v jazyku C++

Modifikátory prístupu

- Podobne ako v Jave, ale bez tzv. package access
- Implicitný prístup v triede je **private**
- Trochu iná syntax

```
class Student {
  private:
    char[20] meno;
    char[20] priezvisko;
    int rocnik;
  protected:
    char* poznamka;
  public:
    void zapis(int rocnik);
    Student(int rocnik);
    ~Student();
  ...
};
```

- Trieda navyše môže sprístupniť súkromné položky funkciám, ktoré deklaruje ako priateľské (`friend`)

Druhy dedenia

- Dedenie v C++ môže byť **public**, **protected** a **private**
- Druhy dedenia v C++ kopírujú modifikátory prístupu vzhľadom na vzťah dedenia (a všetko, čo z neho vyplýva):[3]
  - **public**: vzťah dedenia je známy všetkému kódu, ktorý má prístup k základnej a odvodenej triede
  - **protected**: vzťah dedenia je známy len odvodenej triede a triedam od tejto triedy odvodených
  - **private**: vzťah dedenia je známy len odvodenej triede (inheritance with cancellation)
- Inak povedané, druhom dedenia regulujeme viditeľnosť zdedených prvkov

Dedenie public

- Dedenie **public** je v podstate ako dedenie (**extends**) v Jave

```
class GUtvar {
  ...
};

class Kruh : public GUtvar {
  private:
    int r;
  public:
  ...
};
```

[3] http://stackoverflow.com/questions/860339/difference-between-private-public-and-protected-inheritance-in-c

Virtuálne funkcie

- C++ vyžaduje „zapnutie“ polymorfizmu funkcií
- Dynamické viazanie je časovo náročné – programátor ho takto má pod kontrolou, ale musí stále na to myslieť
- Funkcia, ktorá nie je označená ako virtuálna, sa neprekonáva (len sa prekryje)

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

void A::m() {
   cout << "toto je A" << endl; // štandardný výstup
}

class B : public A {
   public:
      virtual void m();
};

void B::m() {
   cout << "toto je B" << endl;
}

. . .
A* x = new B();
x->m(); // "toto je B"
```

Čisto virtuálne funkcie

- Virtuálne funkcie nemusia vôbec mať implementáciu
- Ak je to deklarované inicializáciou $= 0$, potom ide o tzv. čisto virtuálne funkcie (pure virtual functions)

```
class A {
   public:
      virtual void m() = 0;
};
```

- Také funkcie potom zodpovedajú abstraktným funkciám v Jave
- Trieda, ktorá obsahuje čisto virtuálne funkcie je abstraktná a nemôže mať inštancie
- Dá sa však urobiť inštancia triedy, ktorá obsahuje obyčajné virtuálne funkcie, a ktoré sú bez implementácie[4]

[4] http://www.codeproject.com/useritems/PureVirtualFunctionCall.asp

Dedenie private a protected

- Ak sa neuvedie iný modifikátor prístupu, predpokladá sa dedenie **private**
- Pri dedení **private** trieda zdedí len **public** a **protected** časť a to do vlastnej **private** časti
- Preto sa toto dedenie označuje aj ako *inheritance with cancellation*
- Toto dedenie je podobné agregácii, lebo navonok sa podtrieda nejaví ako podtyp nadtriedy – podtriede však áno
- Triedy odvodené od triedy, ktorá dedila spôsobom **private**, k takto zdedeným prvkom už nemajú prístup
- Dedenie **protected** je podobné, ale ďalšie odvodené triedy majú prístup k prvkom zdedeným z pôvodnej triedy[5]

```
class A {
   public:
      int x;
   protected:
      int y;
   private:
      int z;
};

class B : public A {
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A {
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A {
   // x is private
   // y is private
   // z is not accessible from D
};
```

3 Viacnásobné dedenie

Viacnásobné dedenie

- C++ umožňuje dedenie jednej triedy od viacerých tried – viacnásobné dedenie (multiple inheritance)

```
class A {
   public:
      virtual int aaa();
};
```

[5] http://stackoverflow.com/questions/860339/difference-between-private-public-and-protected-inheritance-in-c/1372858#1372858

```
class B {
   public:
      virtual void bbb(int i);
};
class M : public A, public B {
   ...
   public:
      virtual int aaa();
      virtual void bbb(int i);
   ...
};
```

Použitie viacnásobného dedenia

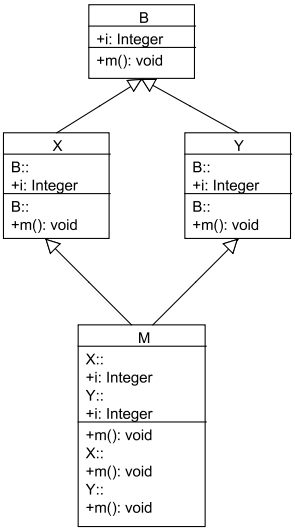
- Aplikujeme vzor Observer na sledovanie času[6]
- Trieda `Clock` je sledovaným predmetom (subject) a musí implementovať operácie na manažment radu objektov, ktoré ju sledujú
- Tieto veci môžeme mať už pripravené v triede `Subject`
- Okrem toho, v iných aplikáciách môžeme potrebovať čistú triedu `Clock`
- Preto `Clock` urobíme zvlášť, a sledovanú verziu vytvoríme pomocou viacnásobného dedenia

```
class ObservedClock : public Clock, public Subject
{
   public:
      virtual void Tick() {
         Clock::Tick();
         Notify();
      }
};
```

- Klientsky kód sa môže na `ObservedClock` pozerať aj ako na `Clock`, aj ako na `Subject`

[6] R. C. Martin. Java and C++: A Critical Comparison. http://www.objectmentor.com/resources/articles/javacpp.pdf

Problém opakovaného dedenia



Dreaded diamond / deadly diamond of derivation

- Problém je v polymorfickom volaní metód
- Predpokladajme, že metóda `m()` je virtuálna

  ```
  B* x = new M();
  ```

- `x->m()` – ktorá metóda sa má vyvolať?
- Kompilátor vôbec nedovolí takéto priradenie – trieda `B` je vo vzťahu dedenia nejednoznačná[7]
- Objekt triedy `M` však môžeme vytvoriť

  ```
  M o;
  ```

- Volanie musí byť plne vymedzené, lebo máme dve metódy `m()`

  ```
  o.X::m();
  ```

- Ale my potrebujeme, aby trieda `M` mala vlastnú implementáciu metódy `m()` a chceme využiť polymorfizmus
- Riešenie je tzv. virtuálne dedenie

Virtuálne dedenie

- Virtuálne dedenie *od spoločnej nadtriedy* zabezpečí, aby jej prvky boli zdedené len raz

[7] http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=167&rl=1

```
class X : virtual public B {
   ...
};
class Y : virtual public B {
   ...
};
class M : public A, public B {
   ...
};
```

- Potom už môžeme zavolať metódu m()

```
B* x = new M();
x->m();
```

4 Šablóny

Šablóny

- Generickosť v Jave je inšpirovaná šablónami (templates) z C++
- Kým je preklad generickosti v Jave riešený zdieľaním kódu uplatnením techniky rušenia typov (type erasure), v C++ sa uplatňuje špecializácia kódu[8]
- Generickosť prináša tzv. parametrický polymorfizmus
- Porovnanie dvoch hodnôt hocijakého typu[9]

```
template <class T> T max(T a, T b) {
   return a>b ? a : b;
}
...
max(10, 15);
```

Metaprogramovanie pomocou šablón

- Šablóny predstavujú jazyk v jazyku
- Template metaprogramming[10]

```
template<int n> struct Factorial {
   enum { RET = Factorial<n-1>::RET * n };
};

template<> struct Factorial<0> {
   enum { RET = 1 };
};

void main() {
   cout << Factorial<7>::RET << endl; // 5040
}
```

[8] http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ100

[9] http://www.cplusplus.com/doc/tutorial/templates/

[10] Krzysztof Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. Thesis, Computer Science Department, Technical University of Ilmenau, Ilmanau, Germany, 1998. Chapter 8. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.202.2797&rep=rep1&type=pdf

5 Preťaženie operátorov

Preťaženie operátorov

- C++ umožňuje preťažiť nielen funkcie, ale aj operátory
- Klasickým príkladom sú komplexné čísla, ktorých podporu C++ má v štandardnej knižnici[11]
- Podobne možno vytvoriť podporu zlomkov
- Nasleduje príklad využitia šablón a preťaženia operátorov[12]

Referencie

- V príklade budu použité *referencie* (iné ako referencie v Jave)

```
int i = 0;
int& r = i;
```

- Bezpečnejšie ako ukazovatele
  - Nedajú sa meniť po inicializácii, ani sa nedajú definovať bez inicializácie
  - Nemôžu byť **null**
  - Používajú sa predovšetkým na prenášanie parametrov a návratovej hodnoty vo funkciách

Referencie a parametre

- Príklad:[13]

```
void duplicate (int& a, int& b, int& c) {
   a *= 2;
   b *= 2;
   c *= 2;
}

int main () {
  int x = 1, y = 3, z = 7;
  duplicate (x, y, z);
  cout << "x = " << x << ", y = " << y << ", z = " << z;
  return 0;
}
// Výstup: x = 2, y = 6, z = 14
```

[11] http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=192&rl=1
[12] I. Polášek. Programovací jazyk C++, prednáška, 2006.
[13] http://www.cplusplus.com/doc/tutorial/functions2/

Referencie a návratová hodnota

- Príklad:[14]

```
int& preinc(int& x) {
   ++x;
   return x;
}

...

   preinc(y) = 5; // 5 bude priradené y
```

Preťaženie operátorov – príklad

- Chceme realizovať operácie nad zlomkami pomocou bežných operátorov – napr. sčítanie pomocou operátora +[15]
- Na zjednodušovanie zlomkov potrebujeme funkciu na výpočet najväčšieho spoločného deliteľa

```
template <class T> T nsd(T i1, T i2) {
   T i;
   while(i2 > 0) {
      i = i1 % i2;
      i1 = i2;
      i2 = i;
   }
   return i1;
}

template <class T> class Zlomok {
   protected:
      T citatel, menovatel;

   public:
      Zlomok& Zjednodus() {
         T NSD = nsd(citatel, menovatel);
         citatel /= NSD;
         menovatel /= NSD;
         return *this;
      }

      Zlomok (T _citatel=0, T _menovatel=1) {
         citatel = _citatel;
         menovatel = _menovatel;
      }

   friend Zlomok<T> operator+(Zlomok<T> &z1, Zlomok<T> &z2);
   friend Zlomok<T> operator*(Zlomok<T> &z1, Zlomok<T> &z2);
   friend ostream& operator<<(ostream& s, Zlomok<T> z);
};

template <class T> ostream& operator<<(ostream& s, Zlomok<T> z) {
   return (s << z.citatel << '/' << z.menovatel );
}
```

[14] http://en.wikipedia.org/wiki/Reference_(C++)
[15] Príklad poskytol Dr. Ivan Polášek.

```
template <class T> Zlomok<T> operator+(Zlomok<T> &z1, Zlomok<T> &z2) {
   return Zlomok<T>(z1.citatel * z2.menovatel + z2.citatel * z1.menovatel,
                    z1.menovatel *z2.menovatel).Zjednodus();
}

template <class T> Zlomok<T> operator*(Zlomok<T> &z1, Zlomok<T> &z2) {
   return Zlomok<T>(z1.citatel * z2.citatel,
                    z1.menovatel * z2.menovatel).Zjednodus();
}

void main() {
   Zlomok<int> a(1,2), b(2,1), c;
   c=a*b;
   cout << c;
   c=a+b;
   cout << c;
}
```

6 Sumarizácia

- C++ je komplexný jazyk
- Zložitý model dedenia – tri typy dedenia podľa modifikátorov prístupu a podpora viacnásobného dedenia
- Neposkytuje rozhrania, ale dajú sa simulovať triedami, ktoré obsahujú výlučne čisto virtuálne funkcie
- Umožňuje riadenie polymorfizmu (`virtual`) – účinnosť a flexibilita sú v rukách programátora
- Nepodporuje priamo garbage collection – dá sa však implementovať, ale v Jave sa nedá vypnúť
- Šablóny – jazyk v jazyku
- C++ zavádza referencie – bezpečnejšie ako ukazovatele, ale iné ako referencie v Jave
- Ďalšie rozdiely medzi Javou a C++ zahŕňajú:
  - výnimky – C++ nemá blok **finally**
  - C++ nemá priamu podporu viacniťovosti
  - vhniezdené typy v C++ sú len na vymedzenie platnosti názvov