Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 15.02. 2001
Url: http://www.builder.cz/art/cpp/cpp_dedicnost.html

Jednoduchá dědičnost v C++

Jednoduchá dědičnost v C++

V tomto článku objasním jednoduchou dědičnost v C++. Obecně pojem dědičnost jsem vysvětlil ve svém článku "Základní pojmy OOP". Jedná se o jeden z možných vztahů mezi třídami. V tomto článku se budu věnovat pouze implementaci v C++. Pro ty z Vás, kterým dědičnost není zcela jasná doporučuji si nejprve přečíst článek "Základní pojmy OOP".

Dědičnost slouží ke rozšiřování vlastností tříd. V tomto článku se budu věnovat jen jednoduché dědičnosti, tedy dědičnosti, kdy nějaká třída dědí (Je speciálním případem.) jen jedné třídy. Jestliže třída B je speciálním případem třídy A, potom říkáme, že třída B dědí z třídy A. Třída B je vlastně třída A, ale má také nějaké vlastnosti navíc. Abychom metody třídy B, které jsou stejné jako metody třídy A, nemuseli psát podruhé, použijeme mechanismus dědění. Mějme například třídu vozidlo, která má soukromý atribut počet kol a veřejné metody které počet kol nastaví a vrátí. Z ní bude dědit třída nákladní vozidlo, která má navíc soukromý atribut nosnost a veřejné metody, které nosnost nastaví a vrátí. Třída nákladních vozidel má tedy také atribut počet kol a metody pro nastavení a vrácení počtu kol. Nemusí se psát znovu.

Možnosti jak dědit

Existují 3 způsoby jak v C++ dědit. Podle způsobu dědění se rozhoduje nakolik budou v potomkovi viditelné (přístupné) metody, nebo atributy předka. Způsoby jsou:

- public
- protected
- private

Jak vidíte, jsou použity stejná klíčová slova, jak pro způsob dědění, tak pro označení přístupových práv k atributům, nebo metodám. To asi proto, aby byl začínající programátor dokonale zmaten. Pro pochopení těchto způsobů uvedu následující tabulku. V horním řádku jsou přístupová práva položky třídy v předkovi. V levém sloupci jsou způsoby jakým byl potomek zděděn.

	Položka v předkovi je private.	Položka v předkovi je protected.	Položka v předkovi je public.

Dědění je private.	Položka v potomkovi není přístupná.	Položka v potomkovi je private.	Položka v potomkovi je private.
Dědění je protected.	Položka v potomkovi není přístupná.	Položka v potomkovi je protected.	Položka v potomkovi je protected.
Dědění je public.	Položka v potomkovi není přístupná.	Položka v potomkovi je protected.	Položka v potomkovi je public.

Položka v potomkovi nemusí být přístupná, ale vždy v potomkovi je. Abych přiznal pravdu, nevím v jakém případě by bylo vhodné použít jinou dědičnost, než veřejnou (public). Tím samozřejmě nechci říci, že dědění private či protected jsou k ničemu, nebo špatné. Já ale budu dále používat jen veřejnou dědičnost. Příklad:

```
#include <iostream.h>

class Vozidlo
{
 private:
    int PocetKol;
 public:
    void nastavKola(int a){ PocetKol = a; }
    int dejPocetKol(){ return PocetKol;}
};

/* Nákladní vozidlo dědí z vozidlo způsobem public. */
class NakladniVozidlo : public Vozidlo
{
 private:
    int Nosnost;
 public:
    void nastavNosnost(int a){ Nosnost = a; }
    int dejNosnost(){ return Nosnost;}
};
```

Instance třídy nákladní vozidlo je zároveň také instancí třídy vozidlo. Opačně to platit nemusí. Dostali jsme se k jedné ze základních vlastností dědičnosti: "na místě, kde je očekáván předek, může být dosazen potomek". Tedy ukazatel, nebo reference na instanci třídy předka může klidně ve skutečnosti ukazovat na instanci potomka. Pro příklad doplním předchozí zdrojový text o funkci main, kde ukážu mnoho možností:

```
int main(void)
{
   Vozidlo *v1 = new Vozidlo;
   Vozidlo *v2 = new NakladniVozidlo; /* Naprosto korektní! */
   NakladniVozidlo *n1 = new NakladniVozidlo;
   v1->nastavKola(4);
   v2->nastavKola(6);
   n1->nastavKola(8); /* Instance n1 je vlastně i instancí třídy Vozidlo */
```

```
    cout << v1->dejPocetKol() << '\t' << v2->dejPocetKol() << '\t' << n1->dejPocetKol() << endl;
    n1->nastavNosnost(1000);
    cout << n1->dejNosnost() << endl;
    delete v1;
    delete v2;
    delete n1;
    return 0;
}
```

Můžete si lehce vyzkoušet, že v těle nějaké metody třídy nákladní vozidlo nelze přímo použít atribut `int PocetKol`, ale lze k němu přistupovat jen pomocí veřejných metod třídy vozidlo. To svědčí o tom, že třída nákladní vozidlo položku `int PocetKol` má, ale není k ní přístup. Položka v předkovi je `private`, dědění je `public` - viz tabulka. Bylo by chybou napsat v třídě nákladní vozidlo atribut `int PocetKol`, protože třída nákladní vozidlo by měla dvě proměnné `int PocetKol`. To by ani tak nevadilo jako fakt, že metody třídy vozidlo by pracovali s jiným atributem `int PocetKol` než metody třídy nákladní vozidlo. Tato chyba ale není syntaktickou chybou, takže na ni překladač v době kompilace neupozorní!

Samozřejmě, že nyní může dále ze třídy nákladní vozidlo dědit další třída, ze které může zase dědit jiná třída atd... Stejně tak může třída vozidlo mít více následovníků než jen nákladní vozidlo - třeba osobní vozidlo atd... Pořád se jedná o jednoduchou dědičnost.

Chování konstruktorů při dědění

Konstruktory (stejně jako destruktory) jsou metody, které se nedědí, ale lze implicitně vyvolat konstruktor předka v konstruktoru potomka. Implicitně při vytváření instance nějaké třídy se nejprve vyvolá bezparametrický konstruktor (Pokud není, tak implicitní konstruktor vytvořený překladačem.) nejvyšší nadtřídy, poté jejího potomka, atd, až se vyvolá požadovaný konstruktor pro instanci. Pro lepší pochopení uvedu příklad. Do výše uvedeného příkladu vepište mezi veřejné metody třídy vozidla konstruktor:

```
Vozidlo():PocetKol(0)
{
   cout << "Tvořím vozidlo" << endl;
}
```

A mezi veřejné metody třídy nákladní vozidlo vepište konstruktor:

```
NakladniVozidlo():Nosnost(0)
{
    cout << "Tvořím nákladní vozidlo" << endl;
}
```

Nyní po spuštění programu bude dobře vidět, jaké je pořadí volání jednotlivých konstruktorů. Mnohdy ale není vhodné volat bezparametrické

konstruktory a poté nastavovat hodnoty. Implicitní volání konstruktorů předků lze potlačit a nahradit jej voláním konstruktorů s nějakými parametry. Provádí se to stejným způsobem jako se volají konstruktory atributů třídy (Viz můj článek "Vytváření instancí, konstruktory destruktory".). Pro příklad doplňte mezi veřejné metody třídy vozidlo konstruktor:

```
Vozidlo(int kola):PocetKol(kola)
{
   cout << "Tvořím vozidlo s parametrem" << kola <<endl;
}
```

Třída Vozidlo má nyní dva konstruktory. Konstruktor třídy nákladní vozidlo upravte následovně:

```
NakladniVozidlo():Vozidlo(6),Nosnost(0)
{
   cout << "Tvořím nákladní vozidlo" << endl;
}
```

Opět spusťte program a zjistíte, že při vytváření instance n1 je zavolán konstruktor Vozidlo(6) a potom konstruktor NakladniVozidlo().

Chování destruktorů při dědění

Destruktory se stejně jako konstruktory nedědí, ale stejně jako u konstruktorů jsou implicitně volány destruktory předků. Na rozdíl od konstruktorů se volají v opačném pořadí. V mém příkladě se při likvidaci instance n1 nejprve vyvolá destruktor ~NakladniVozidlo() a poté destruktor ~Vozidlo(). Můžete se o tom přesvědčit, jestliže definujete v mém příkladě destruktory třídám, které stejně jako konstruktory vypíšou na stdout informaci o tom, že proběhly. Bude možná jen překvapeni, že při likvidaci instance v2 nebude zavolán destruktor ~NakladniVozidlo(), ale jen destruktor ~Vozidlo(). Proč tomu tak je vysvětlím ve svém příštím článku věnovaném virtuálním metodám.

Není-li Vám zcela jasné v jakém pořadí se konstruktory a destruktory volají doporučuji Vám si opsat můj příklad i s doplňky a důkladně jej odkrokovat v debuggeru.

Dodatek

Musím ještě upozornit, že další metoda, která se nedědí, je přetížený operátor = . Přetíženým operátorům jednou věnuji celý článek, kde vše vysvětlím. Toto upozornění jsem napsal jen pro ty, kteří již přetěžování znají od jinud a zkoušeli by přetěžování i dědičnost ve svých programech.

V mém příštím článku se budu věnovat polymorfismu, tedy metodám volajících se tak zvanou pozdní vazbou (jsou deklarovány s klíčovým slovem `virtual`). Potom v dalším článku (spíše v dalších dvou článcích) se vrátím k dědičnosti, tentokráte vícenásobné.

------ http://www.builder.cz ------