Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 28.08. 2001
Url: http://www.builder.cz/art/cpp/cpp_typdedit.html

Problémy s typy při vícenásobné dědičnosti

V minulém článku jsme si ukázali použití operátorů `dynamic_cast, static_cast` a `reinterpret_cast`. Dnes se podíváme na některé záludnosti při přetypování instancí tříd vzniklých vícenásobnou dědičností. Vytvořme si tři jednoduché třídy:

```
#include <iostream>
using namespace std;

class PrvniNadTrida
{
  public:
    virtual void prvni() { cout << "Prvni this=" << this << endl; }
};

class DruhaNadTrida
{
  public:
    virtual void druha() {  cout << "Druha this=" << this << endl; }
};

class PodTrida : public PrvniNadTrida, public DruhaNadTrida
{};
```

První dvě třídy mají po jedné metodě, které vypíšou adresu, na kterou se odkazuje `this`. Třetí třída zdědí obě tyto metody po svých předcích. Podívejme se v jednoduché funkci `main` na první problém.

```
int main()
{
        PodTrida *pod = new PodTrida;
        pod->prvni();
        pod->druha();
        delete PodTrida;
        return 0;
}
```

Při spuštění programu zjistíme, že jeden objekt, na který se odkazuje ukazatel `pod`, má pří volání každé své metody jinou hodnotu implicitního parametru `this`. Problém spočívá v tom, jak je v C++ implementována vícenásobná dědičnost. Objekt, na který se odkazuje ukazatel `pod` jsou vlastně dva objekty (Jeden třídy `PrvníNadTřída`, druhy třídy `DruháNadTřída`.) za sebou. Nebudu se zde zabývat tématem vícenásobné dědičnosti, protože jsem se jím již zabýval ve svých předchozích článcích. Zmíněný problém s `this` není zas tak velkým problémem. Zkrátka každá metoda má "svůj" `this` a vše funguje. S tímto ale úzce souvisí jiný problém, o kterém se zmíním později. Nejprve ale vytvořme ještě dvě další funkce a změňme funkci `main`:

```
void funkce1(DruhaNadTrida *objekt)
{
        cout << "funkce1: parametr " << objekt << endl;
        objekt->druha();
}

void funkce2(PrvniNadTrida *objekt)
{
        cout << "funkce2: parametr " << objekt << endl;
        objekt->prvni();
}

int main()
{
        PodTrida *pod = new PodTrida;
        funkce1(pod);
        funkce2(pod);
        delete PodTrida;
        return 0;
}
```

Vytvořil jsem 2 funkce, které vypíšou adresu, na kterou se odkazuje ukazatel daný jako parametr, a dále v každé funkci se zavolá metoda objektu, na který se parametr funkce odkazuje. Funkce se liší jen typem parametru. Ve funkci `main` vytvořím instanci třídy `PodTřída` a zavolám obě funkce. V souladu s principy dědičnosti "na místě, kde je očekáván předek může být potomek" předám oběma funkcím jako parametr ukazatel na instanci třídy `PodTřída`. Po spuštění zjistíme, že každá funkce dostane jako parametr jiný ukazatel. Překladač při překladu volání funkcí `funkce1`, `funkce2` správně přetypoval parametr na předka. Při přetypování ukazatele (nebo i reference) na instanci třídy vzniklé vícenásobnou dědičností může dojít ke změně samotné adresy, na kterou se ukazatel (reference) odkazuje. Na tento fakt je dobré pamatovat. Je to taková zvláštnost, objekt vlastně ztrácí svou identitu. jak jsem se již v dřívějších článcích zmiňoval, každý objekt má svou identitu, pomocí níž jej lze jednoznačně odlišit od jakéhokoliv jiného objektu. V C++ je tato identita dána paměťovou adresou objektu, jejíž hodnotu máme v implicitním parametru `this`. Nemohou existovat dva různé objekty na stejné adrese. Stejně tak nemůže, i když v uvedených příkladech se děje opak, být jeden objekt na více adresách. Při vícenásobné dědičnosti v C++ tomu tak ale je. Při jednoduché dědičnosti žádný takový problém nenastane. Pozměňme pro ilustraci znovu funkci `main` takto:

```
int main()
{
```

```
        PodTrida *pod = new PodTrida;
        PrvniNadTrida *prvni = pod;
        DruhaNadTrida *druhy = pod;
        if (prvni != druhy)
        {
                cout << "prvni neni druhy" << endl;
        }
        delete PodTrida;
        return 0;
}
```

Vytvořil jsem instanci třídy `PodTřída`, na kterou se odkazuje ukazatel `pod`. Dále jsem vytvořil dva ukazatele, které "ukazují" na stejný objekt, na který "ukazuje" `pod`. Nyní chci porovnat, zda ukazatele `první` a `druhy` jsou stejné (To znamená, zda ukazují na stejný objekt, nebo-li "Je objekt, na který ukazuje ukazatel `první` identický s objektem na který ukazuje ukazatel `druhý`?". Z předchozích dvou řádků plyne, že ano. Přesto po spuštění programu zjistíme, že ne. Právě tato ztráta identity objektu je podle mne velikou nevýhodou vícenásobné dědičnosti v C++. Ještě jen podotknu, že některým překladačům se asi po právu nebude líbit, že porovnáváme ukazatele různých typů. Přesně by mělo porovnání vypadat `if ( (void*) prvni != (void*) druhy)`. Na chování programu to ale nic nezmění. Jak tedy zjistit, zda jsou objekty identické?

K tomuto účelu nám slouží operátor `dynamic_cast`. Používáme-li vícenásobnou dědičnost, můžeme identitu objektů porovnat pomocí "triku" - přetypování na `void*` pomocí operátoru pro dynamické přetypování. Porovnání má správně vypadat

```
if (dynamic_cast<void*>(prvni) == dynamic_cast<void*>(druhy))
{
        cout << "Jsou stejné" << endl;
}
else
{
        cout << "Nejsou stejné" << endl;
}
```

Je třeba si uvědomit, že ve svém předchozím porovnání jsem použil chybné přetypování pomocí operátoru `(typ)`, před kterým jsem v minulém článku varoval. Použití správného operátoru `dynamic_cast` vyřeší náš problém. Při použití vícenásobné dědičnosti totiž nemusí platit rovnost `dynamic_cast<typ*> (ukazatel) == ukazatel`. Na tento fakt je nutné pamatovat. Při použití jednoduché dědičnosti žádný problém s identitou nevzniká.

Tímto jsme ukončili téma přetypování v C++. V příštím článku se budeme věnovat šablonám funkcí, čímž otevřeme velikou kapitolu o šablonách v C++.