Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 25.09. 2001
Url: http://www.builder.cz/art/cpp/vnorsabl.html

Vnitřní typy u parametrů šablon, vnořené šablony v C++

V dnešním článku se budeme věnovat dvěma nesouvisejícím tématům. Prvním z nich bude práce s vnitřními typy u parametrů šablon, s čímž souvisí pro nás nové klíčové slovo `typename`. Jako druhé téma jsem vybral vnořené šablony, tedy šablony, ve kterých je deklarována další šablona.

Vnitřní typy u parametrů šablon v C++

V jazyce C++ může být v jednom typu deklarován typ jiný - vnitřní (vnořený) typ. Vnitřní typ může být jednak třída deklarovaná uvnitř třídy, nebo typ uvnitř třídy deklarovaný pomocí klíčového slova `typedef`. Opět, jako vždy ve svých článcích, se zabývám pouze třídou, vše co zde napíši je obdobně použitelné i pro struktury a unie. Má-li být schopen s vnitřním typem pracovat kdokoliv jiný, než objekt dané třídy, musí být vnitřní typ veřejný. Uveďme si jednoduchý příklad vnitřní třídy:

```
class VnejsiTrida
{ /* Vější třída */
        public:
                class VnitrniTrida
                { /* Vnitřní třída */
                        private:
                                int Atribut;
                        public:
                                int dejAtribut() { return Atribut; }
                                void nastavAtribut(int a) { Atribut = a; }
                };
                VnitrniTrida &dejObsah(); /* Vrátí instanci vnitřní třídy. */
        private:
                VnitrniTrida Obsah; /* Vnější třída má jako atribut instanci
                                       vnitřní třídy. */
};
VnejsiTrida::VnitrniTrida &VnejsiTrida::dejObsah()
{
        return Obsah;
}
```

Nyní pro ukázku jak s vnitřní třídou pracovat předvedu velice jednoduchou funkci `main`.

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

int main(void)
{
        VnejsiTrida vnejsi;
        vnejsi.dejObsah().nastavAtribut(10);
        VnejsiTrida::VnitrniTrida vnitrni = vnejsi.dejObsah();
        cout << vnitrni.dejAtribut() << endl;
        return 0;
}
```

Nyní se vraťme zpět k šablonám. Představme si situaci, kdy chceme vytvořit šablonu, která by jako svůj parametr měla typ. Předpokládali by jsme, že za parametr dosadíme vždy třídu, která bude mít vnitřní typ pojmenovaný jako VnitřníTřída. Tedy například naší třídu. Překladač ale v době kdy "si prohlíží" šablonu neví, že parametr možná bude mít vnitřní třídu. Proto musíme použít klíčové slovo `typename`. Uveďme si jednoduchý příklad.

```
template<class Typ> void vypisObsah(ostream &o, Typ parametr)
{
        typename Typ::VnitrniTrida v; /* v je objekt */
        /* Nestačilo by
                VnitrniTrida v
           protože VnitniTrida není globální typ, a překladač
          jej proto pochopitelně nezná.
        */
        v = parametr.dejObsah();
        o << v.dejAtribut() << endl;
}
```

Celý můj příklad je pouze ukázka použití `typename`. Jinak nedává žádný smysl. Snad jsem tím nikoho nezmátl. Vytvořil jsem šablonu funkce. Parametrem šablony je typ jménem `Typ`. Parametry funkce (parametry instance šablony) jsou výstupní proud a instance typu `Typ`. O typu `Typ` vím, že má vnitřní typ jménem `VnitřníTřída`. Bude-li šablona instanciována s parametrem, který nemá jako veřejný vnitřní typ se jménem `VnitřníTřída`, dojde k chybě. V praxi může existovat hodně tříd, které budou mít vnitřní třídu s názvem `VnitřníTřída`. Každá `VnitřníTřída` bude ale úplně jiná, bude mít jen stejně pojmenované metody, které ale mohou dělat úplně jinou činnost. Díky toho lze vytvářet velmi abstraktní šablony. Například až se budeme věnovat knihovně STL a datovým kontejnerům v C++ uvidíme, že každý kontejner má vnitřní typ jménem `iterátor`. Nyní ještě použití naší šablony.

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

int main(void)
{
```

```
        VnejsiTrida v;
        v.dejObsah().nastavAtribut(10);
        vypisObsah(cout,v);    /* Nebo to samé: */
        vypisObsah<VnejsiTrida>(cout,v);
        fstream soubor("file.txt");
        vypisObsah(soubor,v);
        return 0;
}
```

Vnořené šablony

Nyní se podíváme na situaci, kdy v šabloně je deklarována šablona. Vytvořme si velice jednoduchou šablonu třídy, podobnou šabloně Obal z mého minulého článku. Tato třída bude mít přetížen operátor = jako svou metodu. Budeme ale chtít, aby náš operátor byl použitelný pro mnoho typů, které nemusíme v době psaní naší šablony ani znát.

```
template<class Typ> class Obal
{
        private:
                Typ Atribut;
        public:
                void nastavAtribut(Typ a) { Atribut = a; }
                Typ dejAtribut() { return Atribut; }
                template<class Parametr> Obal<Typ> &operator=(Parametr druhy);
};
```

Vytvořili jsme šablonu Obal. Šablona má jednu vnořenou šablonu - operátor =. Nyní vytvoříme implementaci tohoto operátoru.

```
template<class Typ> template<class Parametr>
Obal<Typ> &Obal<Typ>::operator=(Parametr druhy)
                /* Pozor !! Nelze template<class Typ, class Parametr> */
{
        Atribut = druhy; /* Musí být operátor = pro typy Typ a Parametr */
        return *this;
};
```

Jak jsem již upozornil ve zdrojovém textu, nelze napsat `template<class Typ, class Parametr>` , protože by se nejednalo o vnořenou šablonu, ale o šablonu s dvěma parametry. Nyní si ukážeme použití naší šablony.

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

```
int main(void)
{
        Obal<float> pi; /* Přibližně 22/7 */
        pi = 22.0 / 7; /* Obal<float> operator=(float druhy) */
        cout << pi.dejAtribut() << endl;
        pi = 10; /* Obal<float> operator=(int druhy) */
        cout << pi.dejAtribut() << endl;
        return 0;
}
```

Pro náš obal může být parametrem operátoru = cokoliv, co lze přiřadit pomocí operátoru = (implicitního, nebo přetíženého) k jeho atributu.

Pro dnešek je to všechno. V příštím článku se pokusíme vytvořit celkem i praktickou šablonu. Bude to asi i první zdrojový text v mém seriále, který bude mít snad i nějaké praktické využití. Pokusíme se totiž vytvořit pole, které nebude mít dolní index 0, ale libovolnou programátorem zadanou hodnotu. Například indexovat jej bude možné od -10 do +10, nebo od 50 do 100. Bude se jednat o pole, jaké známe například z programovacího jazyka Pascal. V dalších článcích se potom začneme věnovat knihovně STL.