Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 28.03. 2002
Url: http://www.builder.cz/art/cpp/itercnt.html

Použití čítače referencí v C++

V minulém článku Inteligentní ukazatel - čítač referencí v C++ jsme si implementovali šablonu `Pointer`. Jedná se o "inteligentní" ukazatel, který se odkazuje na objekt s čítačem referencí. Není-li na objekt žádná reference, objekt bude automaticky zlikvidován. Dnes si ukážeme jak s náš `Pointer` používat.

Nejprve si vytvoříme jednoduchou třídu. Instance této třídy budeme používat v následujících příkladech.

```
class Trida
{
 private:
  int Pocet;
 public:
  Trida() : Pocet(0) { cout << "konstruktor" << endl; }
  ~Trida() { cout << "destruktor" << endl; }
  void metoda() { cout << "metoda " << ++Pocet << endl; }
  virtual void constmetoda()
    const { cout << "konstantni metoda" << endl; }
};
```

Nyní vytvoříme "inteligentní" ukazatel na instanci třídy Třída. Parametr šablony je typ prvku, na který "inteligentní" ukazatel ukazuje. Parametrem konstruktoru může být jednak jiný "inteligentní" ukazatel (kopírovací konstruktor), nebo normální ukazatel na vytvořený objekt. Všimněte si také na příkladu, že lze použít i operátory == a !=. V příkladech nesmíme zapomenout, že `Pointer` je deklarován v souboru `pointer.h` v prostoru jmen `www_builder_cz`.

```
#include<iostream>
#include<exception>
#include"pointer.h"

using namespace std;
```

```
// Tady musí být deklarace tříd uvedená na začátku článku !

int main()
{
  /*
     Vytvořím objekt, který předám "inteligentnímu" ukazateli.
     Více se o tento objekt nemusím starat. Bude automaticky
     zlikvidován v momentě, kdy na něj již nebude existovat žádný
     "inteligentní" ukazatel.
  */
  www_builder_cz::Pointer<Trida> ptr(new Trida);
  /* S instancí ptr pracuji jako s normálním ukazatelem. */
  ptr->metoda();
  (*ptr).metoda();
  www_builder_cz::Pointer<Trida> ptr2, ptr3;
  if (ptr2.isNULL())
  {
      cout << "ptr2 je NULL" << endl;
  }
  if (ptr2 == ptr3)
  {
      cout << "Správně" << endl;
  }
  else
  {
      cout << "Špatně" << endl;
  }
  ptr2 = ptr;
  ptr->metoda();
  ptr2->metoda();
  ptr2->constmetoda();
  /*Vytváříím nový objekt. Celkem už jsou vytvořeny 2.*/
  ptr3 = new Trida;

  if (ptr3 != ptr2)
  {
      cout << "Správně" << endl;
  }
  ptr3 = ptr2;
  /*
     Tímto je přepsán poslední odkaz na druhý vytvořený objekt.
     Objekt je likvidován destruktorem.
```

```
   */
   ptr2.setNULL();
   try
   {
       ptr2->metoda();
   }
   catch (std::exception &e)
   {
      cout << e.what() << endl;
   }
   cout << "Konec" << endl;
   return 0;
   /* Nyní bude automaticky volán destruktor prvního objektu. */
}
```

K nastavení ukazatele na `NULL` slouží metoda `setNULL()`. Předáme-li jednou našemu "inteligentnímu" ukazateli normální ukazatel, je dobré odstranit všechny normální ukazatele na tento objekt. Inteligentní ukazatelé jej totiž zlikvidují až se jim to bude zdát vhodné. Na normální ukazatele neberou ohled. Buď budeme s ukazatelem pracovat pomocí normálních ukazatelů nebo pomocí šablony `Pointer`. Kombinovat obě možnosti je velmi nebezpečné.

Inteligentní ukazatel jako parametr funkce, či metody

Instance šablony `Pointer` může být bez problému použita jako typ parametru funkce.

```
void fce(www_builder_cz::Pointer<Trida> poi)
{
  cout << "Ve funkci: " << endl;
  poi->metoda();
  (*poi).constmetoda();
  /*
     Nezapomeňte, že počet referencí je nyní o 1 více.
     Nová reference je "poi".
  */
  cout << "Počet referencí " << poi.getReferenceCount() << endl;
}
```

Možná někoho od používání `Pointer` odradí fakt, že má k dispozici již mnoho funkcí, či metod, které jako své parametry vyžadují normální ukazatele. I tento problém lze řešit. Zde ale již musíme dávat trochu pozor. Začínáme totiž kombinovat přístup k objektu pomocí normálních a našich

"inteligentních" ukazatelů. Předání skutečného ukazatele na objekt jako parametru funkce či metody by to nevadilo pokud:

- Ve funkci či metodě nedojde k dealokaci objektu, na který se odkazuje ukazatel předán jako parametr. Důvod je jasný. Funkce, nebo metoda by zničila objekt, se kterým by "inteligentní" ukazatel dále pracoval.
- Funkce, nebo metoda neudělá s ukazatelem nějaký "vedlejší efekt". Například nezapíše ukazatel na náš objekt jako globální proměnnou, nebo jako atribut objektu, jedná-li se o metodu. Vznikl by tím odkaz na něco, co může být kdykoliv dealokováno.

Představme si například, že máme nějakou funkci `void f(Trida *t);`. Je-li `ptr` typu `Pointer<Trida>`, lze funkci volat `f(&*ptr);`. Tím získáme skutečný (normální) ukazatel na objekt třídy Třída.

Funkce a metody nezávislé na typu ukazatele

Existuje také možnost vytvořit funkce či metody nezávislé na tom, zda se jako parametr použije normální, nebo "inteligentní" ukazatel. jak asi každého napadne, jedná se o šablony, jejichž parametrem je typ ukazatele. Příklad:

```
template<class Pointer> void sablona(const Pointer point)
{
  point->constmetoda();
  (*point).constmetoda();
}
```

Použití:

```
www_builder_cz::Pointer<Trida> ptr(new Trida);
Trida *tr = new Trida;
sablona(ptr);  // Lze i náš ukazatel
sablona(tr);   // Lze i normální ukazatel
delete tr; // tr musíme sami zničit  :-)
```

Přetypování

Problémy s přetypováním jsem nastínil ve svém předchozím článku. Dnes si ukážeme jak metodu `cast` použít. Znovu jen zopakuji, že `cast` je naprosto nepoužitelná pro přetypování "inteligentního" ukazatele na primitivní datový typ. Také není vhodná na přetypování instancí třídy vzniklých vícenásobnou dědičností. Metoda `cast` je vnořená šablona. Typ této šablony je novým typem, na který chceme přetypovat. Ukážeme si vše na příkladu.

```
// "Trida" je deklarována výše
```

```
// Také budeme používat "funkce" a "sablona",
// které jsou již deklarovány

class PodTrida : public Trida
{
 public:
  virtual void constmetoda() const
       { cout << "konstantni metoda podtridy" << endl; }
};

void neco()
{
  www_builder_cz::Pointer<PodTrida> pod(new PodTrida);
  sablona(pod); // Zde se přetypovat nemusí
  /*
     Nelze fce(pod) , protože Pointer<PodTrida>
     není potomkem Pointer<Trida>.
  */
  fce(pod.cast<Trida>());
  www_builder_cz::Pointer<Trida> ptr = pod.cast<Trida>();
  ptr->constmetoda(); // Polymorfismus funguje.
}
```

Pointer NENÍ garbage collector!

Šablona pointer není žádný garbage collector, jaký známe z jiných programovacích jazyků (např. Java). Není to dokonce ani náhrada za garbage collector. Když jsem se v upoutávce na článek o garbage collectoru zmínil, jednalo se spíše o nadsázku. Jak jsem již v minulém článku upozorňoval čítač referencí je bezmocný na kruhové vazby mezi objekty. Máme-li například objekty A,B. V A je ukazatel ukazující na B, v B je ukazatel ukazující na A. Nebude-li existovat žádný jiný ukazatel na A, nebo B, objekty by měly být oba zničeny. Nic takového se ale nestane, protože každý z objektů bude mít čítač referencí nastaven na 1. Takové "kruhy" musíme sami "přetrhnout". Například metodou `setNULL`, kterou zavoláme pro jeden z ukazatelů tvořících kruh. Tento problém by pro skutečné GC například z Javy nebyl žádným problémem. Skutečné GC totiž pracují trochu jinak.

Shrnutí

Šablonu `Pointer` rozhodně doporučuji k prostudování, ale při praktickém použití v programech má nějaké ty nevýhody. Pokud o nich ale víme, a nepřekvapí nás, myslím že je možné "inteligentní" ukazatele používat. Zvlášť za velkou nevýhodu považuji problémy s přetypováním. Ty z Vás, kteří nečtou mé články pravidelně, asi překvapilo, co to vlastně ten identifikátor `Pointer` je. Jedná se o šablonu, která je ke stažení zde. Šablonou `Pointer` jsem se pokusil vylepšit standardní čablonu auto_ptr.

Tímto téma čítače odkazů ukončíme. Příště se podíváme na problém, který s počítáním odkazů jen trochu souvisí. Podíváme se na potlačení kopírování velkých objektů. Představme si, že předáváme jako parametr funkce či metody velký objekt. Tento parametr je nutné zkopírovat, což je u velkých objektů neefektivní. Z nějakých důvodů ale nechceme předávat pouze referenci, nebo ukazatel. Jak to řešit? Uvidíme příště.