Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 22.03. 2002
Url: http://www.builder.cz/art/cpp/ipointer.html

Inteligentní ukazatel - čítač referencí v C++

Ve svém předchozím článku o automatických ukazatelích jsme si povídali o šabloně `auto_ptr`. V závěru článku jsme si pověděli o nevýhodách této šablony. Dnes se ji pokusíme vylepšit.

Našim úkolem je vytvořit něco (inteligentní ukazatel), čemu předáme ukazatel na již existující objekt. Tento ukazatel "zapomeneme" a k objektu budeme přistupovat jen pomocí "inteligentního" ukazatele. Takových ukazatelů na jeden objekt může být více. Nemusíme volat destruktor pro objekt. Destruktor bude automaticky zavolán v momentě, kdy na objekt nebude existovat již žádný ukazatel. Vše zajistíme implementací čítače referencí. Každý objekt bude vědět kolik je na něj ukazatelů. Nejprve si vytvoříme "obal" objektu, na který se budou odkazovat "inteligentní" ukazatele. Bude se jednat o šablonu třídy jménem _ObjectWRC (Object with references counter). Parametrem šablony bude třída, na jejíž objekty se budeme odkazovat. Šablona _ObjectWRC bude zapouzdřovat normální ukazatel na požadovaný objekt a čítač referencí. Čítač referencí je číslo udávající počet odkazů na objekt. Šablona _ObjectWRC je pouze pomocná šablona. Uživatel by s ní neměl nijak přijít do styku. (Proto také podtržítko na začátku.) Implementace této šablony je jednoduchá.

```
template<class Type> class _ObjectWRC
{
 private:
  unsigned int ReferenceCount; // Počet odkazů na objekt
  Type *Data; // Zapouzdřený ukazatel

  // Metody, které nesmí být nikdy zavolány.
  _ObjectWRC(const _ObjectWRC<Type>&);
  _ObjectWRC &operator=(const _ObjectWRC<Type>&);

 public:
  _ObjectWRC(Type *normalPointer)
    : ReferenceCount(0), Data(normalPointer) {}
  ~_ObjectWRC() { delete Data; }

  // Metody pro manipulaci s daty:
  const Type *getData() const { return Data; }
  Type *getData() { return Data; }
```

```
  void setData(Type *normalPointer) { Data = normalPointer; }

  // Metody pro práci s čítačem odkazů
  unsigned int getReferenceCount() const { return ReferenceCount; }
  unsigned int increment() { return ++ReferenceCount; }
  unsigned int decrement() { return --ReferenceCount; }
};
```

Instance třídy _ObjectWRC<něco> se nesmí nijak kopírovat. Proto jsou kopírovací konstruktor a operátor = soukromé. Jako parametr konstruktoru bude "normální" ukazatel na existující objekt. V konstruktoru musíme také vynulovat počet referencí. Dále jsme vytvořili metody set a get pro manipulaci s objektem. K dispozici je také metoda `getReferenceCount`, která vrátí počet referencí na objekt. Tuto službu poskytnou naše "inteligentní" ukazatele programátorovi. Programátor tedy bude vědět, kolik ukazatelů ukazuje na objekt, se kterým pomocí jednoho ukazatele pracuje. Poslední dvě metody slouží ke zvýšení a snížení počtu referencí na objekt.

Inteligentní ukazatel

Nyní vytvoříme šablonu `Pointer`, což bude náš "inteligentní" ukazatel. Šablona bude mít přetížené operátory tak, aby práce s ní nám co nejvíce připomínala práci s normálními ukazateli. Popíšu zde jen ty nejdůležitější metody. Obě šablony jsou kompletní k dispozici na konci článku. Pro implementaci platí pravidla:

- Má-li ukazatel ukazovat na nový objekt, sníží počet referencí na objekt, na který ukazoval dříve o 1.
- Má-li ukazatel ukazovat na nový objekt, zvýší počet referencí na nový objekt o 1.
- Je-li ukazatel likvidován (Například na konci aktuálního bloku.), sníží počet referencí na objekt o 1.
- Vzniká-li ukazatel, zvýší počet referencí na objekt, na který se bude odkazovat o 1.
- Pokaždé, když se snižuje počet referencí, ukazatel zkontroluje, jestli počet referencí již není 0. Jestliže ano, potom objekt zlikviduje destruktorem.

Implementace:

```
template<class Type> class Pointer
{
 private:
  _ObjectWRC<Type> *MyObject; // Objekt s čítačem referencí
 public:
  // Konstruktory
  Pointer() : MyObject(NULL) {}
  Pointer(_ObjectWRC<Type> *object);
  Pointer(const Pointer<Type> &pointer);
```

```
  Pointer(Type *normalPointer);

  // Destruktor
  ~Pointer();

  // Operátory =
  Pointer<Type> &operator=(const Pointer<Type> &pointer);
  Pointer<Type> &operator=(Type *normalPointer);

  // Relační operátory
  bool operator==(const Pointer<Type> &pointer)
  {
    return this->MyObject == pointer.MyObject;
  }
  bool operator!=(const Pointer<Type> &pointer)
  {
    return !(this->MyObject == pointer.MyObject);
  }

  // Operátory dereference
  const Type &operator*() const throw (std::runtime_error);
  Type &operator*() throw (std::runtime_error);

  // Operátory -> (Operátory pro přístup k prvkům objektu)
  const Type *operator->() const throw (std::runtime_error);
  Type *operator->() throw (std::runtime_error);

  // Metoda vracející počet ukazatelů na objekt, na který
  // se odkazuje tento ukazatel. Vždy alespoň 1
  unsigned int getReferenceCount() const
  {
    return MyObject == NULL ? 0 : MyObject->getReferenceCount();
  }

  // Metoda nastaví ukazatel na NULL
  void setNULL();

  // Metoda zjistí, zda-li je ukazatel NULL
  bool isNULL() { return MyObject == NULL; }

  // Metoda pro přetypováni ukazatele
  template<class NewType> Pointer<NewType> cast()
```

```
    const throw(std::bad_cast);
};
```

Parametrem šablony je opět typ prvku, na který se bude odkazovat ukazatel. Atributem třídy je ukazatel na objekt s čítačem referencí. Poměrně jednoduché metody, které jsou `inline` jsou relační operátory != a ==, které porovnávají, zda ukazatele ukazují na stejný objekt. Dále metoda `getReferenceCount` vracející počet ukazatelů na objekt. Protože minimálně jeden ukazatel (ten, jehož metodu voláme) na objekt ukazuje, je výsledek vždy alespoň 1. Podívejme se na kopírovací konstruktor. Při volání kopírovacího konstruktoru vlastně vytváříme nový ukazatel. Musíme zvýšit počet referencí.

```
template<class Type>
Pointer<Type>::Pointer(const Pointer<Type> &pointer)
 : MyObject(pointer.MyObject)
{
  if (MyObject != NULL)
  {
    MyObject->increment();
  }
}
```

Nastavíme si ukazatel a zvýšíme počet referencí. Obdobně vypadají oba operátory =. S tím rozdílem, že ještě také sníží počet referencí u starého objektu. Viz. zdrojový text na konci článku. Odečítání počtu referencí lze nejlépe vidět v destruktoru.

```
template<class Type> Pointer<Type>::~Pointer()
{
  if ( (MyObject != NULL) && (MyObject->decrement() == 0) )
  {
    delete MyObject;
  }
  MyObject = NULL;
}
```

Nemůžu zlikvidovat objekt, je-li na něj alespoň jeden odkaz. Objekt musím zlikvidovat, není-li již na něj žádný odkaz. Jinak by jej už nikdy nešlo zlikvidovat.

Nyní si vysvětlíme jeden z operátorů *. Druhý je v podstatě stejný. Obdobně také vypadají operátory ->. Budeme-li se snažit dereferencovat ukazatel NULL, bude vyvržena vyjimka typu `runtime_error` z prostoru jmen `std`.

```
template<class Type>
const Type &Pointer<Type>::operator*() const throw(std::runtime_error)
{
  if (MyObject == NULL)
  {
    throw std::runtime_error("Null pointer is dereferenced");
  }
  register Type *ret = MyObject->getData();
  if (ret == NULL)
  {
    throw std::runtime_error("Null pointer is dereferenced");
  }
  return *ret;
}
```

Nejprve jsme zjistili, jestli náhodnou není ukazatel NULL. Jestliže ano, je vyvržena vyjimka. Potom jsme pro jistotu zjistili, jestli instance _ObjectWRC<něco> náhodou nezapouzdřuje NULL. Jestliže ano vyvrhneme vyjimku. Jestliže ne, vrátíme referenci na skutečný objekt.

Asi nejzajímavější a nejkontroverznější je vnořená šablona `cast`. Slouží k přetypování ukazatele. Nebýt vnořené šablony `cast`, měla by šablona `Pointer` problémy s přetypováním i s dědičnosti typů. Například máme-li nadtřídu, ze které dědí podtřída, při práci s obyčejnými ukazateli můžeme na místo ukazatele na nadtřídu kdykoliv dosadit ukazatel na podtřídu. Ale místo `Pointer<nadtřída>` stěží dosadíme `Pointer<podtřída>`. Budeme muset použít šablonu `cast`.

```
template<class Type> template<class NewType>
Pointer<NewType> Pointer<Type>::cast() const throw(std::bad_cast)
{
  if (MyObject == NULL)
  {
    return Pointer<NewType>(static_cast<NewType*>(NULL));
  }
  if (dynamic_cast<NewType*> (MyObject->getData()) == NULL)
  {
    throw std::bad_cast();
  }
  return Pointer<NewType>((_ObjectWRC<NewType>*)(MyObject));
}
```

Pomocí `dynamic_cast` zjistím, jestli je přetypování možné. Jestliže ne, vyvrhnu vyjimku `std::bad_cast`. Nakonec použiji úplně obyčejné přetypování ukazatelů. Podíváme-li se podrobněji na `cast`, zjistíme dva nedostatky.

- Budeme-li používat "inteligentní" ukazatele na primitivní datové typy, nebudeme moc naše ukazatele přetypovat. Tento problém by se asi dal vyřešit specializacemi.
- Vnořená šablona `cast` je naprosto nepoužitelná pro přetypování instancí tříd vzniklých vícenásobnou dědičností. Je to velký problém celého čítače referencí. Vůbec nevím jak ho řešit. Nápady přivítám v diskusi pod článkem.

Myslím ale, že tyto dva nedostatky nejsou zas tak zásadní. I s primitivními datovými typy, i s objekty tříd vzniklých vícenásobnou dědičností se pomocí našich "inteligentních" ukazatelů pracovat dá. Nejdou pouze přetypovávat. O různých řešeních těchto problému si povíme v příštím článku.

Dalším velkým problémem čítače odkazů jsou vazby mezi ukazateli, které tvoří kruh. Představme si graf, ve kterém jsou vrcholy (uzly) objekty, a hrany jsou naše ukazatele. Vznikne-li někde v grafu kruh (cyklus, smyčka), je s čítačem referencí trochu problém. Každý objekt v kruhu má jeden odkaz, proto nebude zničen. Ve skutečnosti ale mají být zničeny všechny, protože na celý kruh už odkaz není. O takovém kruhu musíme vědět, a včas (před ztrátou reference na něj) jej přerušit.

Článek je již příliš dlouhý, proto téma dokončíme příště. Povíme si, jak s našimi ukazateli pracovat. Jak je vytvářet a používat. Jak psát třídy, funkce, či metody nezávisle na tom, zda v nich budou používány obyčejné ukazatele, nebo naše "inteligentní" ukazatele. Povíme si také, jak použít naše ukazatele tam, kde se očekávají pouze ukazatele "normální". Ukážeme si praktické použití naší šablony a poukážeme na možné chyby při práci s ní. Zaujal-li Vás nápad čítače referencí, rozhodně si nenechte ujít příští článek. V tom dnešním je jen polovina informací.

Na závěr je tady šablona `Pointer` ke stažení. Jedná se o soubor pointer.h. Šablona je deklarována v prostoru jmen `www_builder_cz`.

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