Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 01.02. 2001
Url: http://www.builder.cz/art/cpp/cpp_kopir.html

Kopírovací konstruktor v C++

V tomto článku popíšu velmi důležitý konstruktor, kopírovací konstruktor, o kterém jsem se v minulém článku nezmínil. Absence kopírovacího konstruktoru je častou chybou a je velmi častou příčinou "pádů" programů, nebo jejich nevysvětlitelného chování. Abych nejlépe ukázal, jak je kopírovací konstruktor důležitý, vymyslel jsem následující příklad:

```
#include <iostream.h>

class Vektor
{
  private:
     int Rozmer;
     int *Slozky;
  public:
     Vektor();
     Vektor(int rozmer);
     ~Vektor();
     void pricti(const Vektor druhyVektor);
     int dejRozmer() const;
     int dejPrvek(int index) const;
     void nastavPrvek(int index, int hodnota);
};

Vektor::Vektor():Rozmer(0),Slozky(NULL)
{}

Vektor::Vektor(int rozmer):Rozmer(rozmer),Slozky(new int[rozmer])
{}

Vektor::~Vektor()
{
   delete[] Slozky;
}

inline int Vektor::dejRozmer() const
{
   return Rozmer;
```

```
}

inline int Vektor::dejPrvek(int index) const
{
  return ((Rozmer > index) && (index >=0))? Slozky[index] : 0;
}

void Vektor::pricti(const Vektor druhyVektor)
{
  if (this->Rozmer != druhyVektor.dejRozmer())
  {
    return ;
  }
  for(register int p = 0; p<this->Rozmer; p++)
  {
    this->Slozky[p] += druhyVektor.dejPrvek(p);
  }
}

void Vektor::nastavPrvek(int index, int hodnota)
{
  if ((Rozmer > index) && (index >=0))
  {
    Slozky[index] = hodnota;
  }
}

int main(void)
{
  Vektor v1(4),v2(4);
  int p;
  for(p = 0; p<4; p++)
  {
    v1.nastavPrvek(p,p);
    v2.nastavPrvek(p,2*p+10);
  }
  v1.pricti(v2);
  v2.nastavPrvek(2,0); /* BUM!!!!!*/
  cout << "Vysledek: ";
  for(p=0; p<4; p++)
    cout << v1.dejPrvek(p) << '\t';
  cout << endl;
  return 0;
}
```

O tom, jak je navržena třída vektor, by se asi dalo diskutovat. Jedná se jen o ilustrační příklad a k tomu je takto navržená třída dobrá. Na první pohled by

se mohlo zdát, že je vše v pořádku. Přesto spustíte-li program, zjistíte, že program spadne vyvoláním metody instance v2 (Tento řádek jsem okomentoval "BUM"). Abych byl přesný, program havaruje na 100% jen pod OS Linux, který má přísnou kontrolu a ochranu přístupu do paměti. Tento program pod Windows xx spadne jen někdy, což jsou velmi nepříjemné chyby, které se velmi obtížně ladí. Pod Windows by se dalo spíše očekávat, že se v programu nepochopitelně mění některé proměnné a podobně. V každém případě se ale jedná o chybu programátora, ne OS, nebo C++. V příkladu, který jsem uvedl, je chyba. V momentě, kdy je vykonáván řádek: `v2.nastavPrvek(2,0);` již není alokováno pole `Slozky` instance v2. Ukazatel složky v instanci v2 ukazuje na nealokovanou paměť ("nikam").
Nyní nejprve vysvětlím, jak je to možné a poté, jak tomu zabránit.

Volání metody, nebo funkce

Problém mého příkladu není na onom kritickém řádku, ale při volání metody: `v1.pricti(v2);`. Dovolil bych si připomenout, jak probíhá volání metod, nebo funkcí. Program nejprve zkopíruje na svůj zásobník všechny parametry funkce. Jedná-li se o metodu, je navíc kopírován i implicitní parametr `this` . Také se na zásobník uloží návratová adresa, ale to nyní není podstatné. Po skončení funkce, nebo metody se uloží do nějakého registru (podle typu) návratová hodnota a předá se řízení na návratovou adresu volajícímu. Volající přečte návratovou hodnotu, "vyčistí" zásobník a pokračuje v činnosti.
Nyní se dobře podívejme na metodu `pricti` a představme si, co se bude dít při jejím zavolání. Nejprve se na zásobník uloží (zkopíruje) instance třídy Vektor, která je při volání dána jako parametr. Instance je vlastně jen "obyčejný" kus paměti v našem případě délky dvou `int`, což je v 32-bitových OS 2*32 bitů (8 bytů). Tím máme vytvořenu kopii parametru. Tato kopie parametru se v těle metody jmenuje `druhyVektor`. Ukazatel `Slozky` v instanci v2 je zkopírován na ukazatel složky v instanci `druhyVektor`. Není nijak zkopírován obsah paměti (V našem případě pole.), na kterou ukazatel ukazoval. Oba ukazatele (v1.Slozky i druhyVektor.Slozky) tedy ukazují na stejné pole. Po ukončení metody a návratu do volajícího (main) je zničena (odebraná ze zásobníku) instance druhyVektor. Likvidace instancí (viz můj minulý článek) provádí destruktor (v našem případě Vektor::~Vektor()) a ten dealokuje pole, na které ukazoval ukazatel druhyVektor.Slozky. Na stejné pole ale ukazoval ukazatel v1.Slozky. Proto ukazatel v1.Slozky neukazuje na alokovanou paměť. Myslím si, že pochopit tuto "drobnost" je dost významné, takže není-li vám něco jasné, doporučuji si tento program podrobně krokovat v debuggeru a hlídat při tom hodnoty jednotlivých ukazatelů.

Kopírování instancí

Druhy kopií

Existují dva druhy kopií. Tak zvaná "plytká" kopie a "hluboká" kopie. Jako kopii instance mám na mysli situaci, kdy vznikne nová instance podobná, nebo stejná originálu. Vznikne nový objekt s novou identitou. Tedy například:

```
Vektor *a,*b;
a = new Vektor(3);
b = a; /* NENÍ kopie !!! */
```

Zde není vytvořená žádná kopie nějaké instance.
"Plytká" kopie je druh kopie, kterou jsem popsal v odstavci "Volání metody, nebo funkce". Jedná se vlastně jen o takovou povrchní kopii, která

nekopíruje "do hloubky". "Plytké" kopírování tedy zkopíruje hodnoty jednotlivých ukazatelů a nestará se o paměť, na kterou ukazovali. O proti tomu "hluboká" kopie zkopíruje vše, i blok paměti, na kterou ukazují jednotlivé ukazatele.

Kopírovací konstruktory v C++

Kopírovací konstruktor, jak asi nikoho nepřekvapí, se stará o vytváření kopií instancí. Není-li programátorem definován kopírovací konstruktor, je použit tak zvaný implicitní kopírovací konstruktor. Implicitní kopírovací konstruktor vytváří vždy plytkou kopii, což někdy je dobré, jindy není. Chceme-li, aby probíhalo kopírování více do hloubky, nebo aby se při kopírování stala nějaká další činnost, musíme kopírovací konstruktor napsat. Kopírovací konstruktor je konstruktor, který má jako svůj parametr konstantní referenci na instanci třídy, ze které má být vytvářena kopie (tedy své). Kopírovací konstruktor se mimo jiné použije při předávání parametrů hodnotou. Nyní dopíšu kopírovací konstruktor třídy Vektor. Mezi veřejné metody třídy dopište jeho deklaraci: `Vektor(const Vektor& druhy);` a dopište jeho tělo:

```
Vektor::Vektor(const Vektor& druhy)
:Rozmer(druhy.dejRozmer()),Slozky(new int[this->Rozmer])
{
  for(register int p = 0; p<this->Rozmer; p++)
  {
    Slozky[p] = druhy.dejPrvek(p);
  }
}
```

Kopírovací konstruktor mohu samozřejmě i sám v programu volat. Například před zavoláním metody `pricti` mohu napsat `Vektor kopie(v1);` resp. `Vektor *ukazatel = new Vektor(v1);` pro vytvoření kopie instance. Opět všem doporučuji si tento příklad dobře prohlédnout v debuggeru.
Vždy si dobře rozmyslete, co se má a co se nemá kopírovat v kopírovacím konstruktoru. Je jasné, že jestliže instance pomocí ukazatelů sdílejí nějakou paměť, nemá smysl v kopírovacím konstruktoru dělat kopii této paměti. Naopak musíte dát pozor, aby nenastal případ jako v mém ukázkovém programu bez kopírovacího konstruktoru. Vše by se dalo shrnout asi do následující rady (spíše nepsaného pravidla): Používá-li instance nějakou "vnější" paměť, měla by VŽDY mít konstruktor, kde ji alokuje, destruktor, kde ji uvolní, a kopírovací konstruktor, kde vytvoří její kopii. Nepište kopírovací konstruktor v případě, že by prováděl stejnou činnost jako implicitní. Implicitní konstruktor vytvořený překladačem bude určitě rychlejší. Aby jsem nevytvořil mylný dojem, že je vše v pořádku, použijte třídu Vektor i s kopírovacím konstruktorem v nějakém programu, kde deklarujete: `Vektor v1(3),v2;`. Poté někde napište `v2 = v1;`. Zjistíte, že operátor přiřazení vytváří plytkou kopii. Toto lze potlačit přetížením tohoto operátoru. Proto by třída používající "vnější" paměť měla mít také přetížený operátor = . Přetěžování operátorů věnuji jeden celý článek, ale bohužel ne hned. Před tím se chci podívat na dědičnost a věci s ní spjaté, což se určitě nevejde do jednoho článku (nejspíš ani ne do dvou, nebo do tří).

Na závěr bych chtěl jen dodat trochu vtipnou radu. Až dostanete pocit, že váš program je správný, ale překladač je špatný (Co si budeme namlouvat, k takovým smělým hypotézám dojde každý programátor mnohokrát za život.), raději zkontrolujte, jestli každá třída má správně definované kopírovací konstruktory a přetížené přiřazovací operátory, než svůj pocit vyslovíte někomu nahlas. :-)

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