Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 13.04. 2001
Url: http://www.builder.cz/art/cpp/cpp_iostream2.html

Přetěžování operátorů << a >> pro datové proudy v C++

Přetěžování operátorů << a >> pro datové proudy v C++

V minulém článku jsme si něco málo ukázali jak pracovat s datovými proudy a něco o objektech `cin ,cout, cerr`. Dnes si ukážeme jak poslat do (resp. přijmout z) datového proudu nejen primitivní datový typ, ale také instanci nějaké třídy. Také si ukážeme jak používat další stream - fstream, tedy datový proud, jehož "cílem" je soubor.

Jak posílat do datových proudů vlastní objekty?

Jedná se o vlastnost, kterou nám již knihovna stdio z jazyka C neposkytuje. Jde právě o jeden ze dvou minule zmiňovaných důvodů proč používat proudy místo funkcí s stdio. Opět použijeme přetížené operátory << a >>. Hlavní problém je, jak je přetížit. V první řadě musím říci, že operátory je dobré (vlastně i nutné!) přetěžovat jako obyčejné funkce, nikoliv jako členské metody. Důvod je jasný. Operátor přetížený jako členská metoda by musel být metodou nějakého proudu. To však autoři knihovny nemohli udělat, protože v době tvorby knihovny pro datové proudy nemohli znát náš datový typ, pro který budeme požadovat operátor `<<`, nebo `>>`. Stejně tak nemůžeme, jak se v OOP v podobných situacích dělá, vytvořit nějakou svou třídu mujstream, které zdědí vše například z `ostream`, a navíc třídu `ostream` rozšíří o operátor `<<`. Problém je v tom, že ze třídy `ostream` dědí mnoho tříd, které by dědily i nadále z `ostream`, ne z mujstream, takže by metodu << nedědily. Ve svém článku "Přetěžování operátoru 1. díl" jsem přesně uvedl, kdy přetěžovat operátory jako funkce a kdy jako metody. Nelze tedy operátory pro přístup k proudům přetížit jako metody. Smíříme se s tímto faktem a ukážeme si, jak přetížit operátor >> pro nějakou jednoduchou třídu:

```
#include <iostream>
class MojeTrida
{
   private:
      int A,B;
   public:
      int dejA() { return A; }
      int dejB() { return B; }
      void nastavA(int a) { A = a; }
      void nastavB(int b) {B = b; }
};
```

```
istream& operator>>(istream &is, MojeTrida &objekt)
{
   int cislo;
   is >> cislo; /* Využiji již přetížených operátorů pro primitivní datové typy.*/
   objekt.nastavA(cislo);
   is >> cislo;
   objekt.nastavB(cislo);
   return is; /* Nezapomeňte na tuto drobnost! Jinak nebudete moc dávat operátory za sebe. */
}
```

Operátory `<< , >>` by měly vracet referenci na proud, na který byly volány. Je to nepsané pravidlo, aby šlo operátory dávat "za sebe".

Náš operátor přistupuje k soukromým atributům třídy MojeTřída pomocí veřejných metod. Operátor `>>` není ale zase tak "cizí" metoda, abychom ji nedovolili přistupovat k soukromým datům. Čtenář, který je zvyklý dodržovat principy OOP, nyní asi trochu pociťuje odpor k nějakému porušení zapouzdření, ale u operátorů `<<` a `>>` se běžně používá. Musíme si uvědomit, že ne ke všem soukromým proměnným musí být přístup přes veřejné metody. řešením je označit funkci jako `friend` (přátelskou). Pro ukázku přetížím operátor `<<` jako přátelskou funkci. Nejprve musím v definici mojí třídy uvést, že funkce `<<` je přátelská. Vepište prosím do třídy MojeTřída na poslední řádek (před };) text: `friend istream& operátor<<(ostream &os,MojeTrida &objekt);`. Tedy deklaraci funkce s klíčovým slovem `friend`. Kromě funkcí mohou být "přátelské" také třídy. Přátelské funkce, nebo metody přátelských tříd mohou přistupovat k soukromým položkám třídy. Nyní stačí dopsat tělo funkce:

```
ostream& operator<<(ostream &os, MojeTrida &objekt)
{
   return (os << objekt.A << objekt.B);
   /* Krátké že? */
}
```

A jednoduchá funkce main:

```
int main(void)
{
   MojeTrida m;
   cout << "Cekam objekt" << endl;
   cin >> m;
   cout << "Objekt byl:" << endl << m;
   return 0;
}
```

Asi Vás hned napadne, co se asi stane, jestliže místo čísla vložíte nějaký znak. Proudy mají tak zvané stavové bity (příznaky), které nastavují podle toho, zda došlo k nějaké chybě, nebo ne. Existují metody, kterými můžete zjistit, který bit je nastaven, tedy jestli se stalo něco špatného, případně co. Nejjednodušší způsob ale je využít přetíženého operátoru přetypování streamu na `void*`. Je-li například `(void*) cin == NULL` došlo k chybě.

Samozřejmě operátor přetypování na `void*` je k dispozici pro všechny proudy. Je-li proud v chybovém stavu, není schopen data posílat, nebo přijímat. Je nutné jej "resetovat" metodou `clear(int = 0)`, která vyčistí chybové stavy. Pro příklad pozměníme funkci main:

```
int main(void)
{
  MojeTrida m;
  cout << "Cekam objekt" << endl;
  cin >> m;
  if (!cin)
  {
    cerr << "Objekt nenacten! " << endl;
  }
  else
  {
    cout << "Objekt byl:" << endl << m;
  }
  return 0;
}
```

Nyní se podívejme na třídy `ifstream` (resp. `ofstream`), které dědí z `istream` (resp. `ostream`). Jedná se o datové proudy, jejichž zdrojem (resp. cílem) je soubor. Velkou výhodou je, že na `ifstream` lze volat operátor <<, protože `ifstream` je (dědí z) `istream`. Konstruktor těchto tříd může mít mnoho parametrů. Lze jej volat s 1 parametrem - jménem souboru. Otevřeme-li takto soubor a budeme-li používat operátory `<<` a `>>`, bude se jednat o textový soubor a o tak zvaný formátovaný vstup a výstup. O jiných možnostech si povíme v dalším článku. Uvedu jednoduchý příklad. Opět změníme funkci main v předchozím příkladu:

```
int main(void)
{
  MojeTrida m;
  cout << "Cekam objekt" << endl;
  cin >> m;
  if (!cin)
  {
    cerr << "Objekt nenacten! " << endl;
  }
  else
  {
    ofstream f("data.txt");
    if (!f)
    {
      cerr << "Nelze otevrit data.txt" << endl;
      return -1;
    }
    f << "Objekt byl:" << endl << m;
```

```
    cout << "Objekt byl:" << endl << m;
  }
  return 0;
}
```

Třída `ofstream` je deklarována v hlavičkovém souboru `fstream`. Nezapomeňte proto na začátek našeho příkladu připsat: `#include<fstream>`

Soubor se uzavře buď při volání destruktoru streamu, nebo pomocí metody `close()`.
Tolik tedy dnes k datovým proudům a k přetěžování operátorů `<<` a `>>`. Příště ukážu několik málo dalších příkladů a také se podíváme na neformátovaný přístup k datovým proudům - tedy na práci s binárními soubory. V následujícím článku potom k paměťovým proudům, čímž téma datových proudů dokončíme.

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