Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 04.12. 2001
Url: http://www.builder.cz/art/cpp/cppfobj.html

Funkční objekty v C++

Funkční objekt je instance třídy, která má jako svou veřejnou metodu operátor (), tedy operátor pro volání funkce. V dnešním článku si ukážeme jak zobecnit funkci, jak používat funkční objekt na místě, kde se očekává ukazatel na funkci. Než se začneme věnovat využití funkčních objektů, zopakujeme si, jak přetížit operátor (). Ukážeme si vytvoření jednoduchého funkčního objektu.

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

class pokus
{
    public:
       int operator()(int parametr);
};

pokus::operator()(int parametr)
{
    cout << "Je volan operator () s parametrem " << parametr << endl;
    return parametr * 2;
}

int main()
{
    pokus objekt;
    objekt(0); // Nebo: objekt.operator()(0);
    cout << objekt(10) << endl;
    return 0;
}
```

Vytvořili jsme si jednoduchou třídu funkčních objektů, třída ma operátor () pro parametr `int`. Instanci této třídy tedy lze použít (zapsat) jako funkci (volání funkce) s parametrem `int`. Jedná se pouze o ukázkový příklad, který nemá snad žádné využití. Než se dostaneme k poněkud užitečnějším funkčním objektům, podívejme se nejprve na jeden problém, který nemá s funkčními objekty nic společného.

Představme si, že chceme vytvořit funkci, která zjistí, kolik prvků v nějakém kontejneru, nebo poli vyhovuje nějaké podmínce. Podmínku v době psaní

naší funkce neznáme. Naše funkce by měla být univerzálně použitelná, pro jakoukoliv podmínku. Jedno z možných řešení je mít jako parametr naší funkce ukazatel na funkci vracející `bool`, která vrátí, zda její parametr odpovídá podmínce. Ukažme si to na příkladu.

```
#include <iostream>
#include <vector>

using namespace std;

bool podminka(int a)
{ /* Je číslo menší, než 3 ? */
    return a < 3;
}

template<class InputIterator>
int pocetPlatnych(InputIterator zacatek, InputIterator konec, bool (*fce)(int))
{
    register int pocet = 0;
    for(InputIterator i = zacatek; i != konec; i++)
    {
         if (fce(*i))
        {
            pocet++;
        }
    }
    return pocet;
}

int main()
{
    vector<int> v;
    int pole[5] = { 1, 2, 3, 4, 5} ;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);
    v.push_back(-20);
    cout << pocetPlatnych(v.begin(), v.end(), podminka) << endl;
    cout << pocetPlatnych(pole, &pole[5], podminka) << endl;
    return 0;
}
```

Naše funkce `početPlatných` může mít jako svůj třetí parametr jakoukoliv funkci (ukazatel na funkci), která má jako parametr `int` a vrací `bool`. Toto řešení má několik nedostatků. Za prvé v momentě, kdy budeme chtít, použít jinou podmínku (Například <5 místo <3.), nebudeme moci použít funkci

`podmínka`, budeme muset napsat novou podmínku. To by se snad dalo ještě obejít pomocí globální proměnné, ale problém by nastal, kdyby jsme chtěli například podmínku 3 < x < 5. Naše funkce `početPlatných` by se už vůbec nedala použít pro pole, nebo kontejner obsahující jiné prvky než `int`. Chceme-li napsat opravdu obecnou funkci `početPlatných`, musíme použít funkční objekty.

Kdyby jsme si chtěli vytvořit nějakou třídu s definovaným operátorem (), a její instanci použít jako parametr naší funkce `početPlatných`, překladač by to odmítl jako nekompatibilitu typů. (Instance třídy není ukazatel na funkci.) Musíme naší šablonu funkce `početPlatných` poněkud zobecnit. Parametrem šablony nebude jenom typ iterátoru, ale další typ, který jsem nazval `Funkce`. Třetí parametr funkce `početPlatných` bude parametr typu `Funkce`. Tím docílíme toho, že parametrem může být "cokoliv". Tělo funkce `početPlatných` nijak měnit nemusíme. Podíváme-li se podrobně na vzniklou šablonu funkce zjistíme, že za typ `Funkce` může být dosazen buď typ funkce s jedním parametrem, která vrací `bool` (V našem případě může vracet i `int`.), nebo třída, která má jako veřejnou metodu operátor () s jedním parametrem, vracejícím `bool`, nebo `int`. Podívejme se na příklad.

```
#include <iostream>
#include <vector>

using namespace std;

class TridaFunkcnichObjektu
{
    private:
        int HorniMez;
    public:
         TridaFunkcnichObjektu() : HorniMez(0) {}
         TridaFunkcnichObjektu(int mez) : HorniMez(mez) {}
        bool operator()(int i) { return i < HorniMez; }
        void nastavMez(int mez) { HorniMez = mez; }
};

bool podminka(int a)
{
     return a < 3;
}

template<class InputIterator, class Funkce>
int pocetPlatnych(InputIterator zacatek, InputIterator konec, Funkce fce)
{
    register int pocet = 0;
    for(InputIterator i = zacatek; i != konec; i++)
    {
        /*
           fce je typu Funkce. Právě kvůli následujícího řádku jsou
           omezení na typ Funkce. Buď se musí jedna o ukazatel na funkci, nebo
           o třídu funkčních objektů - viz text nad příkladem.
        */
```

```
         if (fce(*i))
        {
            pocet++;
        }
    }
    return pocet;
}

int main(int argc, char **argv)
{
    vector<int> v;
    int pole[5] = { 1, 2, 3, 4, 5} ;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);
    v.push_back(-20);
    cout << pocetPlatnych(v.begin(), v.end(), podminka) << endl;
    cout << pocetPlatnych(pole, &pole[5], podminka) << endl;
    cout << "To same:" << endl;
    TridaFunkcnichObjektu funkcniObjekt(3);
    cout << pocetPlatnych(v.begin(), v.end(), funkcniObjekt) << endl;
    cout << pocetPlatnych(pole, &pole[5], funkcniObjekt) << endl;
    funkcniObjekt.nastavMez(5);
    cout << endl << pocetPlatnych(v.begin(), v.end(), funkcniObjekt) << endl;
    cout << pocetPlatnych(pole, &pole[5], funkcniObjekt) << endl;
    return 0;
}
```

Šablona funkce `početPlatných` je nyní hodně obecná. Lze ji použít na jakýkoliv kontejner, obsahující instance jakéhokoliv typu. Podmínka může být libovolná, kterou lze v C++ zapsat. Použití šablony funkce `početPlatných` je znázorněno v jednoduché funkci `main`. Vytvořili jsme vlastně něco jako obecný algoritmus pracující s kontejnery. V knihovně STL je implementováno mnoho algoritmů pracujících s kontejnery. Až se budeme algoritmy v STL zabývat (což by mělo být v článku po následujícím článku), uvidíme, že v STL je i obdoba naší šablony funkce `početPlatných`.

Dnes jsme si předvedli princip funkčních objektů. Ještě než se podíváme na standardní algoritmy v STL, budeme se v příštím článku zabývat funkčními objekty v STL. V STL je také několik standardních funkčních objektů. Funkční objekty se velmi často používají pro práci se standardními algoritmy, ale je možné je používat i u vlastních šablon funkcí, či metod.