Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 25.10. 2001
Url: http://www.builder.cz/art/cpp/cpp_iter.html

Iterátory v C++

Iterátor v C++ je vlastně taková obdoba ukazatelů pro kontejnery. Dříve, než se dostaneme k bližšímu vysvětlení pojmu iterátor, ukážeme si jednoduchý příklad, jak pracovat s obyčejným polem pomocí ukazatelů.

```
int pole[20];
for (int *temp = pole, temp != &pole[20]; temp++)
{
        *temp = 0;
}
```

Tento způsob není v C++ žádná novinka, známe ho z jazyka C. Druhá možnost je v každém kroku cyklu zvýšit nějakou celočíselnou hodnotu, která bude použita jako index v operátoru []. Uvedený příklad by ale měl být přeložen efektivněji, protože zde odpadá výpočet adresy prvku při vyhodnocení operátoru [].

V minulém článku jsme si ukázali šablonu jednorozměrného pole z STL - `vector`. Kdybychom chtěli s nějakým kontejnerem pracovat pomocí ukazatelů jako v našem příkladě s polem, neuspěli by jsme. Máme ale k dispozici iterátory. Iterátor je vlastně něco jako zobecněný ukazatel. U ukazatele operátor ++ způsobí, že ukazatel ukazuje na paměť bezprostředně následující za pamětí, na kterou ukazoval před vyhodnocením tohoto operátoru. V případě pole v našem příkladě se jedná vždy o následující prvek. Naopak operátor ++ u iterátoru způsobí vždy "přechod" na další prvek v kontejneru, nemusí se ale jednat o paměťové místo bezprostředně za předchozím místem. Chování iterátoru je tedy závislé na druhu kontejneru, který procházíme. Pro různé kontejnery jsou iterátory různé (různě implementovány), mají ale všechny stejné rozhraní (se všemi se pracuje stejně). V kontejnerech jsou metody `begin` a `end`. Metoda `begin` vrátí iterátor na první prvek. Metoda `end` vrátí iterátor za poslední prvek. Prvek na který ukazuje iterátor vrácený metodou `end` je neplatný (nealokován), stejně jako je neplatný v příkladu prvek `pole[20]`. Uveďme si obdobu našeho příkladu pro `vector`.

```
#include <vector>
std::vector<int> vektor(20);
for (std::vector<int>::iterator temp = vektor.begin(), temp != vektor.end; temp++)
{
        *temp = 0;
}
```

Iterátor je tedy typ (třída), který má přetíženy operátory tak, aby se s ním pracovalo stejně jako s ukazatelem. Slouží k procházení kontejnerů. Nemusí se ale vždy jednat o nějakou třídu. Například ve velice jednoduchém příkladu kontejneru `Array` v mém článku "Pole s libovolým intervalem indexování" je iterátor vlastně ukazatel. Mohl jsem si dovolit použít ukazatel jako iterátor, protože jsem měl jistotu, že prvky v `Array` budou vždy fyzicky za sebou.

Výhoda iterátorů spočívá v tom, že pomocí iterátorů mohu pracovat s libovolným kontejnerem, aniž bych musel vědět o jaký kontejner se jedná. Lze tedy napsat obecný algoritmus použitelný pro libovolný kontejner. V době kdy píši tento algoritmus nemusím vědět, pro jaký kontejner bude použit. Může být také použit pro více typů kontejnerů bez úpravy. Až probereme kontejnery, podíváme se na standardní algoritmy v STL, které umějí spoustu operací s kontejnery. Přístup ke kontejnerům je pomocí iterátorů.

Kategorie iterátorů

Existuje několik kategorií iterátorů. Zde je jejich přehled:

Název	Český název	Charakteristika
InputIterator, OutputIterator	vstupní (čtecí) a výstupní (zapisovací) iterátory	operátory: -> , * , = , ==, != , ++ Dereferencovaný vstupní iterátor nemůže být l-hodnota. tedy nemůže být "vlevo" při použití operátoru =. Dereferencovaný výstupní iterátor nemůže být p-hodnota. Algoritmy používající tyto iterátory mohou pomocí jednoho iterátoru procházet kontejner pouze jedenkrát.
ForwardIterator	dopředný iterátor	Kombinace vstupního a výstupního iterátoru. Pomocí jednoho iterátoru lze kontejner procházet vícekrát, ale jen v jednom směru.
BidirectionalIterator	obousměrný iterátor	Iterátor, který může kontejner procházet i v opačném směru. Tedy vlastně dopředný iterátor rozšířený o operátor --
RandomaccessIterator	Iterátor s náhodným přístupem	Má v podstatě všechny možnosti jako ukazatel. Jedná se tedy o obousměrný iterátor rozšířený o operátory: + - < > <= >= []

Všechny iterátory dále mají bezparametrický, kopírovací konstruktor a destruktor. Na místě, kde je očekáván iterátor nějaké kategorie, lze dosadit iterátor požadované kategorie, nebo iterátor který je v této tabulce níže. Tedy například místo vstupního iterátoru lze dosadit také iterátor obousměrný bez potřeby nějakých konverzí, nebo přetypování. Iterátor s náhodným přístupem lze tedy dosadit kamkoliv. Za iterátor s náhodným přístupem lze považovat ukazatel na nějaký prvek v poli. Ukazatel lze tedy dosadit místo jakéhokoliv iterátoru. Iterátory kontejneru STL jsou alespoň vstupní, nebo výstupní. Ukážeme si jednoduchý příklad šablony funkce pro výstup do datového proudu:

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

```
using namespace std;

template<class InputIterator>
int vypis(ostream &os, InputIterator zacatek, InputIterator konec)
{
        register int pocetVypsanych = 0;
        for(InputIterator i = zacatek; i != konec; i++)
        {
                os << *i << endl; /* Pro prvek, který bude v kontejneru
                musí být přetížen operátor << */
                pocetVypsanych++;
        }
        return pocetVypsanych;
}
/* K napsání této funkce jsem nepotřeboval vědět
        z čeho budu zapisovat - výhoda iterátorů
        kam budu zapisovat - výhoda datových proudů
*/

int main()
{
        vector<int> vektor(20);
        int pole[20], *uk;
        vector<int>::iterator iterator it;

        for(it = vektor.begin(); it != vektor.end(); it++)
        {
                *it = 20;
        }
        for( uk = pole; uk != &pole[20]; uk++)
        {
                *uk = 10;
        }
        fstream souborovyProud("vystup.txt");
        cout << "Do souboru jsem zapsal:"
                << vypis(souborovyProud,vektor.begin(), vektor.end()) << endl;
        vypis(cout,pole,&pole[20]);
        return 0;
}
```

Parametrem šablony je nějaký typ. Já jsem tento typ nazval InputIterator. Jedná se ale pouze o název typu. Mohl jsem pochopitelně použít úplně jiný název (např. `SchvalneSpatneOutputIterator`) a vše by fungovalo. Já ale jako parametr šablony očekávám iterátor kategorie vstupní iterátor. Je nepsaným pravidlem, iterátory pojmenovávat tak, jak je uvedeno v tabulce ve sloupci název. Až budeme pracovat se standardními algoritmy z STL , nebo s metodami kontejnerů z STL budeme se setkávat s názvy uvedených v této tabulce. Pro přehlednost je dobré dodržovat tyto konvence i u

vlastních šablon funkcí.

Dále je nutno také upozornit na existencí konstantních iterátorů. Konstantní iterátor je obdoba ukazatele na konstantu. Například:

```
vector<int> a(100);
vector<int>::const_iterator = a.begin();
```

Dále také existují adaptéry iterátorů. Adaptér iterátoru přizpůsobí existující iterátor pro trochu jiné použití. Asi nejzajímavějším adaptérem iterátoru je tak zvaný reverzní iterátor. Reverzní iterátor je vlastně adaptován iterátor s náhodným přístupem. Operátor ++ u reverzního iterátoru má stejný význam jako operátor -- u iterátoru s náhodným přístupem a naopak. Reverzní iterátory jsou vhodné především tehdy, chceme-li aby již existující algoritmus pracující s kontejnerem v pořadí od prvního prvku k poslednímu pracoval beze změny s prvky v opačném pořadí. Kontejner vektor vrací reverzní iterátor metodou `rbegin` resp. `rend`.

Další zajímavou možností, o které je vhodné se v souvislosti s iterátory zmínit, jsou iterátory v datových proudech. V datových proudech existují vstupní (ve vstupních datových proudech) a výstupní (ve výstupních datových proudech) iterátory. Znamená to, že některé algoritmy napsané pro kontejnery z STL jsou použitelné i pro datové proudy. Jak je vidět, tak iterátory umožňují opravdu velmi obecnou práci s daty. například šablona funkce `vypiš` v mém příkladu dokáže pomocí proudových iterátorů také pracovat s datovými proudy. Uvedu příklad jak pomocí této šablony vypsat na stdout soubor, který předchozí příklad vytvořil, a také jak tento soubor kopírovat pomocí `vypiš`.

```
// Kromě funkce main je vše stejné jako v mém příkladě s funkcí vypiš
int main()
{
        ifstream vstup("vystup.txt");
        /* Asi trochu nevhodné pojmenování - vystup.txt
        je výstup minulého programu. Je nutné předchozí program nejprve spustit.
        Tak vytvoříme tento soubor. Zde pro zjednodušení nebudu ošetřovat případ,
        že soubor neexistuje. */
        ofstream kopie("kopie.txt");
        istream_iterator<int,ptrdiff_t> proudovyIterator1(vstup), konec1;
        /* Vytvořen konstruktorem bez parametru se jedna o konec souboru, tedy
        iterátor "za poslední prvek v souboru". */
        vypis(cout,proudovyIterator1,konec1);
        vstup.close();
        vstup.open("vystup.txt");
        istream_iterator<int,ptrdiff_t> proudovyIterator2(vstup), konec2;
        vypis(kopie,proudovyIterator2,konec2);
        /* istream_iterator je vstupní iterátor - jeden iterátor = jeden průchod */
        return 0;
}
```

Parametrem šablony `istream_iterator` je jednak typ prvků, které se budou načítat, a také typ pro vzdálenost mezi prvky. Typ pro proměnnou udávající vzdálenost mezi prvky je většinou `ptrdiff_t`, což je s největší pravděpodobností `signed int`. Existuje také výstupní proudový iterátor. Jeho použití je obdobné. V jeho konstruktoru je jako další parametr možnost zadat řetězec, který bude oddělovat jednotlivé prvky v proudu. Při dereferencování proudových iterátorů se použijí operátory << resp. >>.

Tolik tedy k iterátorům. Původně jsem měl v úmyslu v tomto článku dokončit téma šablony `vector`. Bylo by dobré se podívat na metody `erase` a `insert`. Nezmínil jsem se o nich minule, protože jejich pochopení je potřeba vědět něco o iterátorech. Článek by byl ale příliš dlouhý, proto téma `vector` dokončíme příště.