Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 02.11. 2001
Url: http://www.builder.cz/art/cpp/sabliter.html

Šablona vector v C++ a iterátory

V předminulém článku jsme se zabývali šablonou `vector`. Neukázali jsme si ale metody `insert` a `erase`. Právě těmto metodám se dnes budeme věnovat. Také si ukážeme jak efektivně nakopírovat do kontejneru obyčejné pole, nebo obsah jiného kontejneru. A jak přimět kontejner aby pracoval i s instancemi tříd, které nesplňují podmínky pro uložení v kontejneru.

Metoda insert

Metoda `insert` slouží k vkládání prvků do vektoru. V žádném případě se nejedná o nějakou obdobu operátoru [], nebo metody `at`. Metoda `insert` žádný prvek vektoru nepřepíše. Vkládání prvku pomocí `insert` vypadá tak, že všechny prvky od pozice na kterou chci zapisovat až po konec se posunou "doprava", tedy jejich index bude zvýšen o 1. Poté se nový prvek zapíše na novou pozici. Vektor se při této operaci zvětší o jeden prvek. Kdybychom chtěli něco takového provést v poli, museli by jsme velikost pole zvětšit (nejspíše pomocí `realloc`), poté překopírovat část prvků na nové pozice, a nakonec zapsat nový prvek. V kontejneru `vector` je všechna tato činnost již naprogramovaná a ukrytá v metodě `insert`. Metoda `insert` je 3 krát přetížena (existují 3 varianty této metody).

- void insert(iterator pozice, const Typ& novyPrvek = Typ()); První možnost je předat metodě insert dva parametry. První je iterátor určující pozici pro vložení prvku. Druhý parametr je vkládaný prvek. Implicitní hodnota pro druhý parametr je prvek vytvořený bezparametrickým konstruktorem. Znamená to, že zavoláme-li metodu `insert` pouze s parametrem iterátoru, bude na danou pozici vložen prvek vytvořený bezparametrickým konstruktorem. Ještě jen zbývá podotknout, že identifikátor `Typ` je typ prvků, které jsou ve vektoru vloženy. Identifikátor `iterator` je veřejný vnitřní typ daného kontejneru. V našem případě se jedná o kontejner `vector`, ale i další kontejnery mají vnitřní typ `iterator`. Viz moje předchozí články.
- void insert(iterator pozice, size_type n, const Typ& novyPrvek = Typ()); Další možností je předat metodě `insert` navíc číslo udávající počet nových prvků. V tomto případě nebude vložen jeden prvek (`novyPrvek`), ale `n` jeho kopií.
- template<class InputIterator> void insert(iterator pozice, InputIterator zacatek, InputIterator konec); Zde na pozici danou prvním parametrem vložíme prvky, které jsou v jiném kontejneru mezi iterátory `zacatek` a `konec`. Prvek, na který se odkazuje iterátor `zacatek` bude vložen, prvek na který se odkazuje iterátor `konec` již vložen nebude. Pojem vstupní iterátor (`InputIterator`) jsem vysvětlil ve svém předchozím článku Iterátory v C++.

Metoda erase

Oproti metodě `insert` metoda `erase` prvky z kontejneru odebírá. Existují dvě varianty:

- erase(iterator pozice); Zruší prvek daný iterátorem.
- erase(iterator zacatek, iterator konec); Zruší všechny prvky v oblasti dané iterátory `zacatek` a `konec`. Prvek, na který se odkazuje iterátor `zacatek` bude odstraněn, prvek na který se odkazuje iterátor `konec` již odstraněn nebude.

Další věc, o které bych se chtěl v souvislosti s vektorem zmínit je konstruktor, který má dva parametry - vstupní iterátory. Tento konstruktor vytvoří vektor, který bude obsahovat kopie prvků mezi zadanými iterátory jiného kontejneru. Často se tento konstruktor používá pro vytvoření vektoru z obyčejného pole. Vše si podrobně ukážeme v příkladě.

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

using namespace std;

int main()
{
	int pole[10];
	for (int p = 0; p < 10; p++)
	{
		pole[p] = p;
	}
	vector<int> v1(pole,&pole[10]);
	/* Misto iterátoru ukazatel - viz předchozí článek */
	vector<int> v2(v1.begin()+6,v1.end());
	cout << "Velikost v1: " << v1.size() << " velikost v2: "
		 << v2.size() << endl;
	for (vector<int>::iterator p = v1.begin(); p != v1.end(); p++)
	{
		cout << *p << " ";
	}
	cout << endl;
	for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++)
	{
		cout << *p << " ";
	}
	cout << endl;
	v1.insert(v1.begin(),-100); //Vložím před první prvek -100
	v1.insert(v1.end(),3,500); // Vložím na konec 3 krát 500
	v2.insert(v2.begin()+2,v1.begin(),v1.end());//Vložím celý vektor v1 do v2
	cout << "Velikost v1: " << v1.size() << " velikost v2: "
		 << v2.size() << endl;
	for (vector<int>::iterator p = v1.begin(); p != v1.end(); p++)
```

```
        {
                cout << *p << " ";
        }
        cout << endl;
        for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++)
        {
                cout << *p << " ";
        }
        cout << endl;
        v2.erase(v2.begin(), v2.end()); // Mažu všechny prvky ve v2
        cout << "Velikost v2: " << v2.size() << endl;
        while (!v1.empty())
        {
                  cout << "Mažu " << *v1.begin() << " z v1" << endl;
                  v1.erase(v1.begin());
        }
        cout << "Velikost v1: " << v1.size() << endl;
        return 0;
}
```

Kontejner a typ prvků, který neodpovídá podmínkám pro vkládání do kontejneru

Ve svém článku Datové kontejnery v C++ - Úvod do STL jsem uvedl podmínky pro typ, jehož instance chceme do kontejneru ukládat. Právě u metod `insert` a `erase` vidíme, proč je u vkládaného objektu důležitá schopnost sebe kopírovat. Mnohdy ale můžeme pracovat s třídou, která nemá kopírovací konstruktor, resp. operátor = přetížen a implicitní použít nelze. Viz můj článek Kopírovací konstruktor v C++. Také je možné, že kopírovací konstruktor, či operátor = je deklarován jako soukromá položka. Tím autor třídy dal jasně najevo, že kopírování instancí je zakázáno. Jak pracovat s takovými typy v kontejnerech? Řešení ukážu na kontejneru `vector`, protože ten již známe. Tento postup je ale použitelný pro libovolný kontejner.

Celá finta spočívá v tom, že do kontejneru nebudeme ukládat samotné prvky, ale pouze ukazatele na ně. Vytvoříme si ukázkovou třídu, která nesplňuje podmínky pro to, aby její instance byly ukládány do kontejneru.

```
class TridaINT
{
          private:
                           int *prvek;
          public:
                          TridaINT(): prvek(new int) {};
                          TridaINT(int cislo): prvek(new int) {*prvek=cislo;};
                          ~TridaINT() { delete prvek; };
                          int dejPrvek() const { return *prvek; };
                          void nastavPrvek(const int novy) { *prvek = novy; };
};
```

U této třídy nelze korektně použít implicitní kopírovací konstruktor ani operátor= - povede to k pádu programu. Také si můžete vyzkoušet, že k nekorektnímu chování dojde i tehdy, jsou-li instance této třídy vkládány, či odebírány z libovolného kontejneru.

Kontejner ukazatelů

Ukazatel vždy splňuje podmínky pro uložení v kontejneru. Můžeme vytvořit kontejner ukazatelů, musíme mít ale neustále na mysli dvě věci. Jednak nesmíme do kontejneru uložit ukazatel na lokální objekt (lokální zásobník). Všechny objekty, na které ukazující ukazatelé budou v nějakém kontejneru by měly být vytvořeny pomocí `new`. Dále si také musíme uvědomit, že při likvidaci kontejneru (volání jeho destruktoru) budou zlikvidovány jeho prvky - ukazatele, nebudou ale zlikvidovány samotné objekty, na které bylo z kontejneru odkazováno. O zavolání jejich destruktorů se musíme postarat sami. Například:

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

using namespace std;

int main()
{
    vector<TridaINT*> cisla(3);
    cisla[0] = new TridaINT(1);
    cisla[1] = new TridaINT(2);
    cisla[2] = new TridaINT;
    cisla[2]->nastavPrvek(3); /* cisla[2] je ukazatel, proto -> */
    for(vector<TridaINT*>::iterator i = cisla.begin(); i != cisla.end(); i++)
    {
            cout << (*i)->dejPrvek() << endl;
            /*
                  i je iterátor.
                  *i je ukazatel - prvek kontejneru na který se odkazuje iterátor.
                  **i je objekt na který ukazuje ukazatel daný iterátorem
                  (Obdoba ukazatele na ukazatel). Vlastně je to *(i->operator*()).
                  (*i)->  je přístup k metodám a atributům objektu, na
                              který se odkazuje ukazatel *i daný iterátorem i.
                  &*i je skutečná adresa ukazatele *i (ukazatel na ukazatel).
                           Vlastně je to &(i->operator*()).
                           Nedoporučuji nikde používat!
          */
    }
    for(vector<TridaINT*>::iterator i = cisla.begin(); i != cisla.end(); i++)
    {
     delete *i; /* V tomto případě musíme provést sami! */
    }
    return 0;
```

```
}
```

Tolik tedy k vektoru. V mnoha případech je použití vektoru pohodlnější, než použití pole. Program pracující s vektorem je sice trochu pomalejší (oproti programu pracujícím s polem), ale velkého zpoždění se bát nemusíme. Žádná metoda žádného kontejneru není volána pozdní vazbou, a některé jsou dokonce překládány jako `inline`. Také je fakt, že kontejnery jsou šablony, tedy značná část operací se provede již v době kompilace, při běhu programu se na rychlosti nijak neprojeví. Příště se podíváme na kontejnery `map` a `multimap`, což jsou asociativní pole.