Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 07.01. 2002
Url: http://www.builder.cz/art/cpp/stdalgoritmy.html

Úvod do standardních algoritmů v C++

Dnes se seznámíme se standardními algoritmy v jazyce C++. Standardní algoritmy jsou součástí STL. Jedná se o spoustu užitečných šablon funkcí, které za nás naprogramoval někdo jiný. Na nás je jen, abychom je používali. Také si ukážeme příklady algoritmů pro naplnění kontejneru.

Standardní algoritmy jsou šablony funkcí. Všechny šablony jsou deklarovány v hlavičkovém souboru `algorithm` v prostoru jmen `std`. Algoritmy ve většině případů slouží k práci s kontejnery. Žádný s algoritmů ale nepracuje s kontejnery přímo. K prvkům kontejneru přistupují algoritmy zásadně pomocí iterátorů. Viz můj článek Iterátory v C++. To umožňuje, aby algoritmy byly obecné a jednou napsaný algoritmus byl použitelný pro jakýkoliv kontejner. Díky toho lze algoritmy aplikovat také na obyčejné pole.

S jistým typem algoritmů jsme se již setkali. Jednalo se o množinové operace pro průnik, sjednocení, množinový rozdíl, symetrickou diferenci a určení zda se jedná o podmnožinu. Těmito operacemi se dále nebudeme zabývat. Probrali jsme je v článku Množina a multimnožina v C++.

Algoritmy nepracující s kontejnery

Jak jsem uvedl, většina algoritmů je pro manipulaci s daty v kontejnerech. Jako příklad algoritmů, které s kontejnery nemanipulují uvedu algoritmy `min`, `max`, `swap`. Algoritmus `swap` "vymění" obsah obou prvků. Šablona `swap` má parametr udávající typ obou prvků. Parametrem funkce jsou dvě reference na proměnné, jejíž hodnoty chceme vyměnit. Typ, který je parametrem šablony musí být schopen se kopírovat pomocí kopírovacího konstruktoru a operátoru =. Nelze-li použít implicitní, musí je definovat programátor. Algoritmy `min` a `max` vracejí minimální a maximální prvek. Pro každý algoritmus existují dvě varianty:

- `template <class T>const T& min(const T& a, const T& b);` - parametrem šablony je typ obou prvků a typ návratové hodnoty. Algoritmus vrátí minimální prvek z `a`,`b`. Použije při tom operátor <.
- template <class T, class Compare>const T& min(const T& a, const T& b, Compare comp); - obdobně jako minulá možnost. K výběru minima nebude použit operátor <, ale funkční objekt zadaný jako poslední parametr. Viz můj článek Funkční objekty v C++. Tím zajistíme, aby objekt a a objekt b byly porovnány pomocí jiné, námi vybrané relace.

Pro algoritmus `max` je situace obdobná. Použití `min`, `max` a `swap` je velmi jednoduché.

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

less<int> l;

int main()
{
    int a = 9, b = 10, c = 11;
    cout << min(a,b) << endl;
    cout << max(b,c) << endl;
    swap(c,a);
    cout << min(a,b,l) << endl;
    /*
       Jenom ukázka možností. V tomto konkrétním
       případě jsem chování min pomocí objektu l
       nijak nezměnil.
    */
    return 0;
}
```

Algoritmy pro vyplňování kontejnerů

V C++ existují 4 algoritmy, kterými lze vyplňovat kontejnery nějakými hodnotami. Pro vyplnění kontejneru konstantní hodnotou slouží algoritmy `fill` a `fill_n`. Šablona `fill` má dva parametry. Typ iterátoru (Očekává se dopřední iterátor. Viz můj článek Iterátory v C++.) Druhým parametrem je typ vkládaného prvku. Parametry funkce jsou dva iterátory (začátek a konec) a prvek, který se má vložit mezi prvky dané iterátory začátek a konec. Šablona `fill_n` má 3 parametry. Typ iterátoru (očekává se výstupní iterátor), typ parametru udávající počet vložených prvků a typ vkládaného prvku. Parametry funkce jsou iterátor, udávající začátek, počet prvků a vkládaný prvek. Použití je jednoduché. Ukážeme si jej na jednoduchém příkladu.

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

using namespace std;

int main()
{
    vector<int> vektor(10,3);
    /* vektor má 10 prvků - samé trojky. */
    for(vector<int>::iterator i = vektor.begin();
        i != vektor.end(); i++)
```

```
    {
        cout << *i << '\t';
    }
    cout << endl;
    fill(vektor.begin(),vektor.end(),20);
    /* V celém vektoru jsou dvacítky. */
    for(vector<int>::iterator i = vektor.begin();
        i != vektor.end(); i++)
    {
        cout << *i << '\t';
    }
    cout << endl;
    fill_n(vektor.begin() + 2,5,100);
    /* Do prvku vektor[2] a 4 následujících (celkem 5)
       vloží číslo 100. */
    for(vector<int>::iterator i = vektor.begin();
        i != vektor.end(); i++)
    {
        cout << *i << '\t';
    }
    cout << endl;
    ofstream soubor("Pokus.txt");
    fill_n(ostream_iterator<int>(soubor,","),10,0);
    /*
       Nezapomeňme, že také proudy mají své iterátory.
       Předchozí řádek zapíše do textového souboru "Pokus.txt"
       deset nul oddělených čárkou.
    */
    soubor << endl;
    return 0;
}
```

Pro vyplnění kontejneru nekonstantní hodnotou jsou k dispozici algoritmy `generate` a `generate_n`. Parametry jsou obdobné jako u `fill`, resp. `fill_n` jen s tím rozdílem, že místo parametru šablon udávající typ vkládaného prvku je parametrem třída funkčního objektu (generátoru). A jako parametr funkce místo hodnoty, která se má vkládat, je parametrem funkční objekt (generátor), nebo ukazatel na funkci. Očekává se ukazatel na funkci, která nemá parametry a vrací typ prvku v kontejneru, nebo funkční objekt jehož třída má přetížen operátor () tak, aby vracel typ prvku v kontejneru a neměl parametry. Viz. můj článek Funkční objekty v C++. Uveďme si příklad.

```
#include<iostream>
#include<fstream>
#include<algorithm>
#include<vector>
#include<stdlib.h>
```

```
using namespace std;

class Faktorial
{
    private:
        int I, Vysledek;
    public:
        Faktorial() : I(0), Vysledek(1) {}
        int operator() () { return Vysledek *=++I; }
        /*
           N-tý prvek ve vektoru bude n! (n - faktoriál).
           Upozorňuji, že prvek s indexem 0 je první,
           nikoliv nultý.
        */
};

class Rada
{
    private:
         int A,B;
    public:
         Rada() : A(0),B(1) {}
        int operator() () { swap(A,B); return A += B; }
        /*
           První dva prvky jsou jedničky, každý další prvek
           je součet předchozích dvou. Může se to zdát trochu
           zvláštně napsané, ale pracuje to tak, jak má.
        */
};

int main()
{
    vector<int> vektor(10); /* vektor má 10 prvků*/
    Faktorial f;
    Rada r;
    generate(vektor.begin(),vektor.end(),rand);
    /*
       Jako generátor je ukazatel na funkci rand
       vracející náhodné číslo. Vyplnil jsem vektor
       náhodnými čísly.
    */
    for(vector<int>::iterator i = vektor.begin();
        i != vektor.end(); i++)
    {
        cout << *i << '\t';
    }
    cout << endl;
```

```
    generate(vektor.begin(),vektor.end(),f);
    for(vector<int>::iterator i = vektor.begin();
        i != vektor.end(); i++)
    {
        cout << *i << '\t';
    }
    cout << endl;
    generate(vektor.begin(),vektor.end(),r);
    for(vector<int>::iterator i = vektor.begin();
        i != vektor.end(); i++)
    {
        cout << *i << '\t';
    }
    cout << endl;
    /* Nesmíme zapomenout, že vše lze použít i na pole. */
    int pole[20];
    generate_n(pole,20,r);
    for(int *p = pole; p != &pole[20]; p++)
    {
        cout << *p << '\t';
    }
    cout << endl;
    return 0;
}
```

Jako poslední parametr funkce `generate` je možné zadat ukazatel na funkci (V mém příkladě `rand`.), nebo funkční objekt (V mém příkladě objekt `f` třídy Faktorial, nebo objekt `r` třídy Rada.).

Příště se podíváme na kopírovací a přesouvací algoritmy.