Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 18.01. 2002
Url: http://www.builder.cz/art/cpp/copyalg.html

Kopírovací a přesouvací algoritmy v C++

V minulém článku jsme se dověděli, že standardní knihovna jazyka C++ má k dispozici mnoho užitečných šablon funkcí. Mnoho z nich manipuluje s kontejnery pomocí iterátorů (Viz můj článek "Iterátory v C++"). Dnes se podrobněji podíváme na kopírovací a přesouvací algoritmy.

Kopírovací algoritmy v C++

Ke kopírování dat slouží algoritmy `copy` a `copy_backward`. Parametry šablony `copy` jsou typy vstupních a výstupních iterátorů. Parametry funkce jsou iterátory na začátek a konec zdrojové oblasti. Třetím parametrem je iterátor na začátek cílové oblasti. Deklarace je následující:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator zacatek, InputIterator konec, OutputIterator cil)
```

Algoritmus zkopíruje data počínaje prvkem, na který se odkazuje `zacatek` a konče prvkem daným iterátorem `konec` na pozici danou iterátorem `cil`. Funkce vrací iterátor na poslední zkopírovaný prvek. Algoritmus `copy_backward` je obdobný. S tím rozdílem, že parametry šablony jsou obousměrné iterátory (Viz můj článek Iterátory v C++.), a data se kopírují "od zadu". Je dobré si uvědomit, že všude, kde je možné použit iterátor, lze použít také ukazatel. A také datové proudy mají iterátory. Nyní jednoduchý příklad.

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

using namespace std;

int main()
{
    vector<int> vektor(10);
    set<int> m; /* Viz článek Množina v C++ */
    int pole[10];
    ofstream f("Pokus.txt");
    generate(vektor.begin(),vektor.end(),rand); /* Viz předchozí článek */
    copy(vektor.begin(),vektor.end(),pole);
    copy(vektor.begin(),vektor.end(),ostream_iterator<int>(cout,","));
```

```
    cout << endl;
    fill(pole,&pole[10],0); /* Viz předchozí článek */
    copy(pole,&pole[10],ostream_iterator<int>(cout,","));
    cout << endl;
    copy_backward(vektor.begin(),vektor.end(),&pole[10]);
    copy(pole,&pole[10],ostream_iterator<int>(cout,","));
    copy(pole,&pole[10],ostream_iterator<int>(f,"\n"));
    if (f)
    {
         cout << endl << "Byl vytvořen soubor Pokus.txt" << endl;
    }
    else
    {
        cout << endl << "Problémy se souborem Pokus.txt" << endl;
    }
    insert_iterator<set<int> > i(m,m.begin());
    copy(pole,&pole[10],i);
    copy(m.begin(),m.end(),ostream_iterator<int>(cout," "));
    return 0;
}
```

Postupně kopírujeme data mezi vektorem a polem. Mezi vektorem a datovým proudem `cout`, mezi polem a proudem `cout` atd... Tímto příkladem se můžeme také přesvědčit, že kopírovat lze mezi libovolnými kontejnery jako vektor, nebo množina (Viz můj článek "Množina a multimnožina v C++"). Ale také mezi datovými proudy, nebo poli. Pokud budeme kopírovat prvky na standardní výstup pomocí iterátoru datového proudu, narazíme na jednu nepříjemnost. Oddělovací znak bude vložen i za poslední číslo. To může být někdy nežádoucí (například zde, když na stdout je za posledním číslem znak ','), jindy zase ano (například v souboru Pokus.txt je dobré mít za posledním číslem nový řádek).

Přesouvací algoritmy v C++

Pod pojmem přesun si každý představí smazání dat v jednom kontejneru, a vytvoření těchto dat v kontejneru jiném. Problém je v tom, že takto se algoritmy, které chci popsat nechovají. Slovo "přesouvací" je můj dosti nevhodný překlad slova remove. Nenapadá mne ale žádné jiné slovo, kterým by se dala činnost "remove" algoritmů popsat. Algoritmy `remove` a `remove_if` prvky z kontejneru odstraní. Data nebudou nikam přesunuta. Zde bych chápal význam slova remove jako odstranit. Ale algoritmy `remove_copy` a `remove_copy_if` ani nepřesouvají ani neodstraňují. Nezabývejme se ale názvem článku, a pojďme se podívat na činnost těchto algoritmů. Deklarace šablon:

- `template <class InputIterator, class OutputIterator, class Typ> OutputIterator remove_copy (InputIterator zacatek, InputIterator konec, OutputIterator vysledek, const Typ& hodnota)` - V oblasti dané iterátory `zacatek` a `konec` vybere všechny prvky, které NEjsou rovny s prvkem hodnota. Porovnání bude provedeno operátorem != , který je buď implicitní, nebo jej musíme přetížit. Tyto prvky zkopíruje do jiného kontejneru na pozici danou iterátorem `vysledek`. Původní kontejner se nijak nezmění.
- `template <class InputIterator, class OutputIterator, class TFunkcniObjekt> OutputIterator remove_copy_if (InputIterator zacatek,`

`InputIterator konec, OutputIterator vysledek, TFunkcniObjekt podminka)` - Vybere v oblasti dané iterátory `zacatek` a `konec` všechny prvky, které NEvyhovují podmínce (NEplatí `podminka(*i) == true`, kde i je iterátor v rozsahu `zacatek` až `konec`). Tyto prvky zkopíruje do kontejneru na pozici danou iterátorem `vysledek`. Původní kontejner se nijak nezmění.

- `template <class ForwardIterator, class Typ> ForwardIterator remove (ForwardIterator zacatek, ForwardIterator konec, const Typ &hodnota);` - Obdobný algoritmus jako `remove_copy` s tím rozdílem, že vybrané prvky budou z kontejneru odstraněny. Nebudou nikam přesunuty ani zkopírovány. Odstraněny budou prvky, které jsou si rovny s daným prvkem.
- `template <class ForwardIterator, class TFunkcniObjekt> ForwardIterator remove_if (ForwardIterator zacatek, ForwardIterator konec, TFunkcniObjekt podminka);` - Obdobný algoritmus jako `remove_copy_if` s tím rozdílem, že vybrané prvky budou z kontejneru odstraněny. Nebudou nikam přesunuty ani zkopírovány. Odstraněny budou prvky, pro které platí podmínka.

Algoritmy `remove` a `remove_if` nijak nemění velikost kontejneru. Jejich návratová hodnota je iterátor na poslední platný prvek v kontejneru po provedení algoritmu. Je vhodné zbytek kontejneru (od tohoto iterátoru až po konec) smazat. NEZAPOMEŇTE na to! Je to zdrojem častých chyb. Vše je znázorněno v příkladu. V příkladu je použita šablona třídy `unary_compose`, která slouží ke "skládání" funkcí, nebo funkčních objektů. Parametry šablony jsou dva typy unárních funkcí, nebo třídy unárních funkčních objektů. Konstruktoru předám dva ukazatele na unární funkci, nebo unární funkční objekty. Výsledkem volání operátoru () instanci třídy `unary_compose` je volání O1(O2(x)), kde x je parametr operátoru (), O1 a O2 jsou funkce, nebo funkční objekty dané konstruktoru. Příklad:

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

using namespace std;

int main(int argc, char **argv)
{
    vector<int> v1(10), v2;
    generate(v1.begin(), v1.end(), rand);
    cout << "Původní:" << endl;
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,","));
    cout << endl;
    cout << "Jen čísla menší než 5000:" << endl;
    vector<int>::iterator i =
        remove_if(v1.begin(),v1.end(),bind1st(less<int>(),5000));
         /* Odstranil jsem prvky x, pro které platí 5000 < x */
    v1.erase(i,v1.end()); /* NEZAPOMEŇTE ! */
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,","));
    /*
        Všimněte si, jak jsem "poskládal" podmínku dělitelnosti
        deseti jen ze standardních prostředků.
```

```
    */
    unary_compose<binder1st<not_equal_to<int> >, binder2nd<modulus<int> > >
        delitelne_10(bind1st(not_equal_to<int>(),0),bind2nd(modulus<int>(),10));
    remove_copy_if(v1.begin(),v1.end(),inserter(v2,v2.begin()),delitelne_10);
    cout << endl << "Nic jsem neodstranil:" << endl;
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,","));
    cout << endl << "Jen čísla dělitelná deseti:" << endl;
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,","));
    cout << endl;
    if (!v2.empty())
    {
         int c = v2[0];
        cout <<  "Odeberu všechny čísla " << c << endl;
        i = remove(v2.begin(),v2.end(),c);
        v2.erase(i,v2.end()); /* NEZAPOMEŇTE ! */
        copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,","));
        cout << endl;
    }
    return 0;
}
```

Všimněte si, jak jsem pouze pomocí standardních prostředků C++ "složil" funkční objekt `delitelne_10`. Použil jsem při tom standardní funkční objekty.

Příště se podíváme na vyhledávácí algoritmy.