CUPRINS

Emanuela Cerchez, Marinel Şerban

PROGRAMAREA ÎN LIMBAJUL C/C++ PENTRU LICEU

●●●●

Programare orientată pe obiecte
şi programare generică cu STL

POLIROM
2013

Algoritmii numerici

Algoritmii numerici realizează prelucrări ale unor secvențe numerice. Pentru a utiliza această categorie de algoritmi trebuie să includem fișierul antet `numeric`.

Algoritm	Efect
`accumulate`	Implicit, însumează elementele dintr-un anumit domeniu. Dacă este specificată, execută o altă operație în loc de adunare cu elementele din domeniu
`adjacent_difference`	Implicit, determină șirul diferențelor dintre elementele consecutive ale unui domeniu. Dacă este specificată, execută o altă operație în loc de scădere cu elementele din domeniu
`inner_product`	Implicit, determină suma produselor dintre elementele corespondente a două domenii. Este posibilă specificarea altor operații în loc de sumă și produs
`partial_sum`	Implicit, calculează șirul sumelor parțiale ale elementelor unui domeniu. Este posibilă specificarea unei alte operații în loc de adunare
`iota`[STL]	Construiește o secvență crescătoare de valori consecutive, începând cu o valoare specificată

4.9. Clasa șablon `pair`

Deoarece apare frecvent necesitatea de a lucra cu o pereche de valori, a fost predefinită în biblioteca *STL* clasa șablon `pair`. Pentru a utiliza clasa `pair` trebuie să includem fișierul antet `utility`.

Declarația clasei `pair`

```
template <class T1, class T2> struct pair;
```

Observație

În declarație am utilizat cuvântul-cheie `struct`. În *C++* tipul `struct` este similar cu tipul `class`, diferența constând în faptul că toți membrii tipului `struct` sunt publici.

Definiția clasei `pair`

```
template <class T1, class T2>struct pair
{//date membre
 T1 first;
 T2 second;
 //constructor implicit
 pair(): first(T1()), second(T2()) {}
 //constructor cu doi parametri
 pair(const T1& a, const T2& b): first(a), second(b) {}
 //constructor de copiere cu conversie implicita
 template<class U,class V> pair(const pair<U,V>& p):
  first(p.first), second(p.second) {}
};
```

Observații

1. Clasa `pair` conține două date membre publice: `first` și `second`, având tipul `T1`, respectiv `T2`.
2. Clasa `pair` are doi constructori (constructorul implicit și constructorul de inițializare care construiește o pereche pe baza a două valori specificate ca parametru) și un constructor de copiere.
3. Deși nu sunt explicit definiți în cadrul clasei, compilatorul generează automat destructorul și supraîncarcă operatorul de atribuire.
4. Începând cu *C++ 11*, în clasa `pair` a fost inclusă funcția membră `swap`, care interschimbă `first` cu `second`.

Funcția `make_pair()`

Funcția șablon `make_pair` permite construirea unei perechi pe baza a două valori specificate ca parametru, astfel:

```
template <class T1,class T2>
pair<T1,T2> make_pair (T1 x, T2 y)
{ return ( pair<T1,T2>(x,y) ); }
```

Operatorii de egalitate și operatorii relaționali

Operatorii de egalitate și operatorii relaționali sunt supraîncărcați pentru a funcționa cu perechi. Vom detalia doar operatorul `==` și operatorul `<`, deoarece semnificația celorlalți operatori poate fi descrisă cu ajutorul acestora.

Operatorul de egalitate `==`

```
template <class T1, class T2>
bool operator==(const pair<T1,T2>& a,const pair<T1,T2>& b);
```

Perechea `a` este egală cu perechea `b` (`a==b`) dacă `a.first==b.first` și `a.second==b.second`.

Operatorul relațional `<`

```
template <class T1, class T2>
bool operator<(const pair<T1,T2>& a,const pair<T1,T2>& b);
```

Compararea a două perechi se realizează din punct de vedere lexicografic. Mai exact, perechea `a` este mai mică decât perechea `b` (`a<b`) dacă `a.first<b.first` sau `a.first==b.first` și `a.second<b.second`.

Exemplu

În exemplul următor vom demonstra modul de utilizare a constructorilor, a funcției `make_pair()` și a operatorilor relaționali, folosind perechi în care primul element (`first`) este de tip `string`, iar al doilea (`second`) este de tip `float`. Perechea `p1` este creată utilizând constructorul de inițializare, iar perechea `p2` este creată utilizând constructorul implicit, apoi este inițializată apelând funcția `make_pair()`.

```
#include <iostream>
#include <utility>
using namespace std;
int main()
{pair<string, float> p1("Ionescu Dan", 9.75);
 pair<string, float> p2;
 p2=make_pair("Popescu Ana", 9.43);
 cout<<p1.first<<' '<<p1.second<<'\n';
 cout<<p2.first<<' '<<p2.second<<'\n';
 if (p1==p2) cout<<"Perechi egale\n";
     else cout<<"Perechi neegale\n";
 if (p1<p2) cout<<"p1<p2";
    else cout<<"p1>=p2";
 p1=make_pair("Popescu Ana", 9.98);
 if (p1<p2) cout<<" Dupa schimbare: p1<p2\n";
    else cout<<" Dupa schimbare: p1>=p2\n";
 return 0;
}
```

După execuția acestui program, pe ecran se va afișa:

```
Ionescu Dan 9.75
Popescu Ana 9.43
Perechi neegale
p1<p2 Dupa schimbare: p1>=p2
```

4.10. Alocatorii

Alocatorii sunt clase care definesc modul de alocare a memoriei pentru containerele din *STL*.

4.11. Aplicații

În capitolele următoare vom descrie diferite clase container și vom face aplicații cu acestea. Aici vom demonstra modul de funcționare a unor algoritmi din *STL* utilizând un tablou unidimensional (vector din C). În loc de iteratori vom utiliza pointeri. Acest lucru este posibil deoarece se realizează o conversie implicită a pointerilor în iteratori. Pentru exemplificare, luăm în considerare următoarele declarații:

```
int a[]={10,11,12,13,14,15,16,17,18,19};
int n=10;
```

Inversarea elementelor într-un tablou unidimensional

Declarația funcției `reverse()`:

```
template <class Bidirectionalliterator>
void reverse (Bidirectionalliterator prim,
              Bidirectionalliterator ultim);
```

Funcția `reverse()` inversează ordinea elementelor din domeniul `[prim, ultim)`. Pentru a inversa întregul vector, putem apela funcția `reverse()` astfel:

```
reverse(a,a+n);
```

După inversare, vectorul `a` este: `{19,18,17,16,15,14,13,12,11,10}`.

Pentru a inversa doar elementele din vector de la poziția `st` (inclusiv) până la poziția `dr` (exclusiv), putem apela funcția `reverse()` astfel:

```
reverse (a+st,a+dr);
```

Dacă `st=3` și `dr=7`, după inversare vectorul `a` este:
`{10,11,12,16,15,14,13,17,18,19}`.

Eliminarea elementelor care îndeplinesc o anumită condiție

Declarația funcției `remove_if()`:

```
template <class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if (ForwardIterator prim,
                           ForwardIterator ultim,
                           UnaryPredicate pred);
```

Funcția `remove_if()` elimină din domeniul `[prim,ultim)` toate elementele pentru care predicatul returnează valoarea `true`, păstrând ordinea elementelor neeliminate. Funcția returnează un iterator care indică noua poziție de sfârșit a domeniului obținut după eliminare.

Funcția parcurge domeniul și, la fiecare poziție pe care se află un element care îndeplinește condiția, caută primul element următor care nu îndeplinește condiția și îl plasează pe poziția curentă. Astfel, ordinea elementelor se păstrează, iar algoritmul de eliminare este liniar.

Să presupunem că dorim să eliminăm din vectorul `a` elementele pare. Vom construi o funcție cu rezultat `bool` care să verifice condiția dorită (această funcție va fi utilizată ca predicat).

```
bool EstePar(int x) {return x%2==0;}
```

Pentru a elimina elementele pare din întregul vector `a`, putem apela funcția `remove_if()` astfel:

```
int * pf=remove_if(a,a+n,EstePar);
```

Evident, funcția `remove_if()` nu modifică dimensiunea vectorului `a`. Prin urmare, vom actualiza după apel valoarea variabilei `n` (numărul de elemente din vectorul `a`), scăzând din pointerul `pf` în care am reținut noua poziție de sfârșit a domeniului, pointerul `a` care indică începutul acestuia:

```
n=pf-a;
```

Dacă dorim să realizăm eliminarea elementelor pare doar de pe un interval din vectorul `a`, să spunem intervalul `[a+st, a+dr)`, atunci trebuie să avem grijă să mutăm la stânga cu `nr` poziții toate elementele din intervalul `[a+dr,a+n)` (unde cu `nr` am notat numărul de elemente eliminate).

```
int st=3, dr=6, nr;
int * pnf=remove_if(a+st,a+dr,EstePar);
nr=a+dr-pnf;          //determin numarul de elemente eliminate
copy(a+dr,a+n,pnf);//copiez restul vectorului
n-=nr;                //actualizez dimensiunea vectorului
```

Transformarea şir

Ne propunem să aplicăm o prelucrare tuturor elementelor unui şir de caractere (în cazul nostru, dorim să transformăm toate literele mici din şir în majuscule). Pentru aceasta am putea utiliza algoritmul `for_each()` din *STL*.

```
template <class InputIterator, class Function>
Function for_each (InputIterator prim, InputIterator ultim,
                   Function fn);
```

Algoritmul `foreach()` aplică tuturor elementelor din domeniul `[prim, ultim)` o prelucrare descrisă de funcţia `fn`.

```
#include <iostream>
#include <algorithm>
using namespace std;
string s;
void majuscula(char& c)
{ if (c>='a' && c<='z') c=c-'a'+'A'; }
int main()
{cin>>s;
 for_each(s.begin(), s.end(), majuscula);
 cout<<s;
 return 0; }
```

Observaţie

Funcţiei `for_each` i-am transmis ca parametri doi iteratori (`s.begin()` returnează un iterator care indică începutul şirul, iar `s.end()` returnează un iterator care indică poziţia de după sfârşitul şirului) şi numele funcţiei care realizează prelucrarea (`majuscula`).

Sortarea unui tablou unidimensional

Sortarea elementelor unui container se poate realiza cu ajutorul funcţiei `sort()`. Această funcţie este supraîncărcată:

```
template <class RandomAccessIterator>
void  sort(RandomAccessIterator  prim,  RandomAccessIterator
ultim);
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator prim, RandomAccessIterator ultim,
      Compare comp);
```

Prima formă se poate utiliza pentru a sorta elementele din domeniul [prim,ultim) în ordine crescătoare (considerând pentru compararea elementelor operatorul <, care trebuie să fie definit pentru tipul elementelor containerului).

A doua formă permite specificarea funcției care se utilizează pentru compararea elementelor containerului (printr-un functor sau un pointer de funcție). Aceasta trebuie să fie o funcție cu doi parametri și trebuie să returneze un rezultat de tip bool. Rezultatul va fi true dacă primul parametru trebuie să fie plasat în vectorul sortat înaintea celui de-al doilea parametru.

Metoda de sortare utilizată este un algoritm creat de David Musser în 1997 denumit *introsort* (sau *introspective sort* – sortare introspectivă). Acest algoritm optimizează metoda de sortare *quicksort*, utilizând *heapsort*, pentru cazul în care adâncimea recursiei este prea mare. Complexitatea acestui algoritm este de O(n log n) în cazul cel mai defavorabil (unde cu n am notat numărul de elemente din domeniul ce trebuie sortat).

Sortarea crescătoare a unui tablou unidimensional de numere întregi

Pentru a sorta crescător întreg vectorul:

```
sort(a,a+n);
```

Pentru a sorta crescător doar domeniul [a+st,a+dr):

```
sort(a+st,a+dr);
```

Sortarea crescătoare a unui tablou unidimensional de fracții

Revenim la clasa fractie, pe care am definit-o în capitolul al doilea. Pentru această clasă am supraîncărcat și operatorul <. Sortarea unui tablou unidimensional cu elemente de tip fractie se poate realiza astfel:

```
#include <iostream>
#include <algorithm>
#include "fractie.h"
using namespace std;
fractie a[1000];
int n;
void Citeste()
{cin>>n;
 for (int i=0; i<n; i++) cin>>a[i]; }
void Scrie()
{for (int i=0; i<n; i++) cout<<a[i]<<' ';
 cout<<'\n'; }

int main()
{Citeste();
 sort(a,a+n);
 Scrie();
 return 0; }
```