Algorytmy w STL

TeamBit -Krzysztof Rudlicki

Treść

Artykuły

Przypisy

Licencje artykułu

Algorytmy w STL

Wstęp

Cóż znaczą biblioteki bez <algorithm>? Na pewno mniej, ponieważ każde modyfikacje na wektorach czy ciągach znaków są bardziej uciążliwe i wymagają od użytkownika dodatkowego wkładu pracy na napisanie algorytmu do wykonania określonego problemu. Weźmy pod uwagę przykładowo problem sortowania. Poniżej przedstawiona jest funkcja sortująca bąbelkowo *n*-elementową tablicę 1-wymiarową.

```
void sortowanie_babelkowe(int tab[], int n)
{
  for (int j=n-1; j>0; --j)
    for (int i=0; i<j; ++i)
      if (tab[i]>tab[i+1])
      {
        int temp=tab[i];
        tab[i]=tab[i+1];
        tab[i+1]=temp;
      }
}
```

Kod nie jest długi, ale wygodniej jest napisać:

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

Lista funkcji zawartych w bibliotece <algorithm>

- accumulate()
- adjacent_difference()
- adjacent_find()
- binary_search()
- copy()
- copy_backward()
- count()
- count_if()
- equal()
- equal_range()
- fill()
- fill_n()
- find()
- find_end()
- find_first_of()
- find_if()
- for_each()
- generate()
- generate_n()
- includes()
- inner_product()
- inplace_merge()
- is_heap()
- lexicographical_compare()
- lower_bound()
- make_heap()
- max()
- max_element()
- merge()
- min()
- min_element()
- mismatch()
- next_permutation()
- nth_element()
- partial_sort()
- partial_sort_copy()
- partial_sum()
- partition()
- pop_heap()
- prev_permutation()
- push_heap()
- random_shuffle()
- remove()
- remove_copy()
- remove_copy_if()
- remove_if()
- replace_copy()
- replace_copy_if()
- replace_if()
- reverse()
- reverse_copy()
- rotate()
- rotate_copy()
- search()
- search_n()
- set_difference()
- set_intersection()
- set_symmetric_difference()
- set_union()
- sort()
- sort_heap()
- stable_partition()
- stable_sort()
- swap()
- swap_ranges()
- transform()
- unique()
- unique_copy()
- upper_bound()

- iter_swap()
- replace()

Lista tematyczna

Operacje niemodyfikujące

- for_each — wykonuje operację na każdym elemencie ciągu
- count — liczy ilość wystąpień danej wartości w ciągu
- count_if — zlicza w ciągu ilość wystąpień wartości spełniających warunek
- equal — określa czy dwa zbiory elementów są takie same

Operacje niemodyfikujące - szukanie

- mismatch — znajduje pierwszą parę różnych elementów dwóch ciągów
- find — znajduje pierwsze wystąpienie wartości w ciągu
- find_if — znajduje w ciągu pierwsze wystąpienie wartości spełniającej warunek
- find_end — znajduje ostatnie wystąpienie ciągu jako podciągu
- find_first_of — znajduje jakikolwiek element ze zbioru w danym ciągu
- adjacent_find — znajduje sąsiadującą parę wartości
- search — znajduje pierwsze wystąpienie ciągu jako podciągu
- search_n — znajduje pierwsze wystąpienie ciągu n-tu wartości w ciągu

Operacje modyfikujące

- copy — kopiuje elementy jednego ciągu do drugiego
- copy_backward — kopiuje ciąg do drugiego ciągu, wstawiając elementy na jego koniec
- fill — zastępuje elementy ciągu podaną wartością
- fill_n — zastępuje n elementów ciągu podaną wartością
- generate — zastępuje elementy ciągu wartościami będącymi wynikiem funkcji
- generate_n — zastępuje n elementów ciągu wartościami będącymi wynikiem funkcji
- transform — wykonuje podaną funkcję dla argumentów ze zbioru i zapisuje wyniki w nowym ciągu
- remove — usuwa elementy o podanej wartości
- remove_if — usuwa elementy spełniające warunek
- remove_copy — kopiuje ciąg, usuwając elementy o podanej wartości
- remove_copy_if — kopiuje ciąg, usuwając elementy spełniające warunek
- replace — zastępuje elementy o danej wartości inną wartością
- replace_if — zastępuje elementy spełniające warunek
- replace_copy — kopiuje ciąg, zastępując elementy o danej wartości inną wartością
- replace_copy_if — kopiuje ciąg, zastępując elementy spełniające warunek

Operacje zmieniające kolejność

- partition — umieszcza elementy spełniające warunek przed tymi które go nie spełniają
- stable_partition — umieszcza elementy spełniające warunek przed tymi które go nie spełniają, zachowuje wzajemną kolejność
- random_shuffle — w losowy sposób zmienia kolejność elementów ciągu
- reverse — odwraca kolejność elementów w ciągu
- reverse_copy — kopiuje ciąg, odwracając kolejność elementów
- rotate — dokonuje rotacji elementów ciągu
- rotate_copy — kopiuje ciąg, przesuwając elementy (rotacja)
- unique — usuwa powtórzenia, w taki sposób że wśród sąsiadujących elementów nie ma dwóch takich samych
- unique_copy — kopiuje ciąg, w taki sposób że wśród sąsiadujących elementów nie ma dwóch takich samych
- swap — zamienia ze sobą dwa elementy
- swap_ranges — zamienia ze sobą dwa zbiory elementów
- iter_swap — zamienia ze sobą dwa elementy wskazywane przez iteratory

Operacje sortujące

- sort — sortuje ciąg rosnąco
- partial_sort — sortuje pierwsze N najmniejszych elementów w ciągu
- partial_sort_copy — tworzy kopię N najmniejszych elementów ciągu
- stable_sort — sortuje ciąg zachowując wzajemną kolejność dla równych elementów
- nth_element — ciąg jest podzielony na dwie nieposortowane części elementów mniejszych i większych od wybranego elementu

Operacje wyszukiwania binarnego

Operacje na posortowanych ciągach

- lower_bound — zwraca iterator do pierwszego elementu równego lub większego od podanego
- upper_bound — zwraca iterator do pierwszego elementu większego od podanego
- binary_search — stwierdza czy element występuje w ciągu
- equal_range — zwraca parę określającą przedział wewnątrz którego występuje dana wartość (lub ich ciąg).

Operacje na zbiorze

Operacje na posortowanych ciągach

- merge — łączy dwa posortowane ciągi
- inplace_merge — łączy dwie posortowane części ciągu
- includes — zwraca prawdę jeśli pierwszy ciąg jest podciągiem drugiego
- set_difference — tworzy różnicę dwóch zbiorów
- set_intersection — tworzy przecięcie dwóch zbiorów
- set_symmetric_difference — tworzy zbiór złożony z elementów występujących w tylko jednym z dwóch ciągów
- set_union — tworzy sumę zbiorów

Operacje na kopcu

- is_heap — zwraca prawdę jeśli ciąg tworzy kopiec
- make_heap — przekształca ciąg elementów tak aby tworzyły kopiec
- push_heap — dodaje element do kopca
- pop_heap — usuwa element ze szczytu kopca
- sort_heap — przekształca ciąg o strukturze kopca w ciąg posortowany

Operacje min max

- max — zwraca większy z dwoch elementów
- max_element — zwraca największy z elementów w ciągu
- min — zwraca mniejszy z elementów
- min_element — zwraca najmniejszy z elementów w ciągu
- lexicographical_compare — sprawdza czy jeden ciąg poprzedza leksykograficznie drugi ciąg
- next_permutation — przekształca ciąg elementów w leksykograficznie następną permutację
- prev_permutation — przekształca ciąg elementów w leksykograficznie poprzedzającą permutację

Operacje numeryczne

Zdefiniowane w nagłówku <numeric>

- accumulate — sumuje ciąg elementów
- inner_product — oblicza iloczyn skalarny na elementach dwóch ciągów
- adjacent_difference — oblicza różnice pomiędzy sąsiadującymi elementami w ciągu
- partial_sum — oblicza sumy częściowe ciągu elementów

Operacje niemodyfikujące

for_each()

```
for_each( iterator początek, iterator koniec, funkcja )
```

Działanie

wykonuje operację na każdym elemencie ciągu.

Przykład

poniższy program wywołuje dla każdego elementu dodanego do vectora funkcję echo.

```
void echo(int num)
{
   cout << num;
}

int main()
{
   vector<short> vect;

   vect.push_back(5);
   vect.push_back(4);
   vect.push_back(3);
```

```
   for_each(vect.begin(), vect.end(), echo);
}
```

Na wyjściu pojawi się: 543. Dlaczego? Ponieważ program nie oddzielił liczb żadnymi znakami - na przykład spacjami - przez co się złączą w jedną liczbę.

count()

```
count( iterator początek, iterator koniec, wartość )
```

Działanie

liczy ilość wystąpień danej wartości w ciągu.

Przykład

program przekształca tablicę liczb w wektor i zlicza ilość znajdujących się w nim dwójek.

```
int main()
{
   int tablica[] = { 2, 5, 7, 9, 2, 9, 2 };
   vector<int> v(tablica, tablica+7);
   cout << "Ilosc dwojek w tablicy: " << count( v.begin(), v.end(), 2 );
}
```

count_if()

```
count_if( iterator początek, iterator koniec, funkcja f )
```

Działanie

liczy w ciągu ilość wystąpień wartości spełniających warunek

Przykład

program przekształca tablicę liczb w wektor i zlicza ilość znajdujących się w nim liczb parzystych.

```
bool czyParzysta(int n){
  return (n%2 ? false : true);
}

int main(){
  int tablica[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  vector<int> v(tablica, tablica+10);
  cout << "Ilosc parzystych: " << count_if(v.begin(), v.end(), czyParzysta);
}
```

equal()

```
bool equal( iterator początek, iterator koniec, iterator początek_drugiego )
bool equal( iterator początek, iterator koniec, iterator początek_drugiego, funkcja_porownująca )
```

Działanie

porównywany jest pierwszy zakres elementów (początek, koniec) z drugim zakresem (zaczynającym się w początek_drugiego).

Przykład

program porównuje łańcuch *str* z *napis* oraz z *napis2*. Porównanie ogranicza się do 2 znaków (długość *str*).

```
int main()
{
   string str= "bc";
   string napis= "abcde";
   string napis2= "bcde";

   if( equal(str.begin(), str.end(), napis.begin()) )
      cout << "Takie same\n";
   else
      cout << "Rozne\n";

   if( equal(str.begin(), str.end(), napis2.begin()) )
      cout << "Takie same";
   else
      cout << "Rozne";
}
```

Wynikiem jest kolejno *Rozne* oraz *Takie same*.

Operacje niemodyfikujące - szukanie

mismatch()

```
pair<> mismatch( iterator1 początek, iterator1 koniec, iterator2 początek_drugi )
pair<> mismatch( iterator1 początek, iterator1 koniec, iterator2 początek_drugi, funkcja )
```

znajduje pierwsze różne elementy dwóch ciągów

Wartość zwracana

para znalezionych różniących się wartości, typu *pair<iterator1, iterator2>*

Działanie

porównuje kolejne elementy w dwóch zbiory (pierwszy określony przez *początek, koniec* oraz drugi zaczynający się w *początek_drugi*). Zwraca pierwsze wystąpienie dwóch różnych elementów, lub parę <*koniec*, odpowiedni_koniec_drugi> jeśli nie znajdzie.

find()

```
iterator find( iterator początek, iterator koniec, wartość )
```

Działanie

znajduje pierwsze wystąpienie wartości w ciągu i zwraca iterator do niej, lub jeśli nie zostanie znaleziona, zwraca iterator *koniec*.

Przykład

program tworzy tablicę liczb 0, 10, 20, ..., 90 i sprawdza, czy znajdują się w nim liczby 30 i 33.

```
int main()
{
   vector<int> zbior;
   vector<int>::iterator wynik1, wynik2;

   for( int i = 0; i < 10; ++i )
      zbior.push_back(i*10);

   wynik1 = find(zbior.begin(), zbior.end(), 30);
   wynik2 = find(zbior.begin(), zbior.end(), 33);
   if( wynik1 != zbior.end() )
      cout << "Znaleziono 30.";
   if( wynik2 != zbior.end() )
      cout << "Znaleziono 33.";
}
```

Wynikiem jest: *Znaleziono 30.*

find_if()

```
iterator find_if( iterator początek, iterator koniec, funkcja  )
```

Działanie

znajduje w ciągu pierwsze wystąpienie wartości spełniającej warunek

find_end()

```
iterator find_end( iterator początek, iterator koniec, iterator początek_szukany, iterator koniec_szukany )
iterator find_end( iterator początek, iterator koniec, iterator początek_szukany, iterator koniec_szukany, funkcja )
```

Działanie

znajduje ostatnie wystąpienie ciągu (początek_szukany, koniec_szukany) jako podciągu w przedziale (początek, koniec). Można dostarczyć własną funkcję do porównywania elementów.

find_first_of()

```
iterator find_first_of( iterator początek, iterator koniec, iterator początek_zbiór, iterator koniec_zbiór)
iterator find_first_of( iterator początek, iterator koniec, iterator początek_zbiór, iterator koniec_zbiór, funkcja )
```

Działanie

znajduje choć jeden element ze zbioru (początek_zbiór, koniec_zbiór) w podanym ciągu (początek, koniec). Można dostarczyć własną funkcję do porównywania elementów.

adjacent_find()

```
iterator adjacent_find( iterator początek, iterator koniec )
iterator adjacent_find( iterator początek, iterator koniec, funkcja )
```

Działanie

porównuje kolejne wartości w obu ciągach, aż znajdzie parę tych samych - zwraca wówczas iterator do tej wartości, lub *koniec* jeśli nie ma pary identycznych wartości. Można dostarczyć własną funkcję do porównywania elementów.

search()

```
iterator search( iterator początek, iterator koniec, iterator początek_szukany, iterator koniec_szukany )
iterator search( iterator początek, iterator koniec, iterator początek_szukany, iterator koniec_szukany, funkcja )
```

Działanie

znajduje pierwsze wystąpienie ciągu (początek_szukany, koniec_szukany) jako podciągu w przedziale (początek, koniec). Można dostarczyć własną funkcję do porównywania elementów.

search_n()

```
iterator search_n( iterator początek, iterator koniec, n, wartość )
iterator search_n( iterator początek, iterator koniec, n, wartość, funkcja )
```

Działanie

znajduje pierwsze wystąpienie ciągu złożonego z *n*-tu wartości jako podciągu w przedziale (początek, koniec). Można dostarczyć własną funkcję do porównywania elementów.

Operacje modyfikujące

copy()

```
copy( iterator początek, iterator koniec, iterator początek_kopia )
```

Działanie

kopiuje ciąg (początek, koniec) do drugiego ciągu

Przykład

tworzy ciąg 10 liczb, po czym kopiuje je do drugiego (pustego) ciągu

```
int main()
{
   vector<int> ciag;
   for (int i = 0; i < 10; i++)
      ciag.push_back(i);

   vector<int> kopia(10);

   copy( ciag.begin(), ciag.end(), kopia.begin() );
}
```

copy_backward()

```
copy_backward( iterator początek, iterator koniec, iterator koniec_kopia )
```

Działanie

kopiuje ciąg (początek, koniec) do drugiego ciągu, tak aby kończyły się razem z jego końcem

Przykład

tworzy ciąg 10 liczb, po czym kopiuje je do drugiego ciągu o długości 13 (elementy będą przesunięte, aby wyrównać do końca)

```
int main()
{
   vector<int> ciag;
   for( int i = 0; i < 10; i++ )
      ciag.push_back(i);

   vector<int> kopia(13);
```

```
   copy_backward( ciag.begin(), ciag.end(), kopia.end() );

   for( int i = 0; i < kopia.size(); i++)
      cout << kopia[i] << " ";
}
```

Wynikiem będzie *0 0 0 0 1 2 3 4 5 6 7 8 9*.

fill()

```
fill( iterator początek, iterator koniec, wartość )
```

Działanie

zastępuje elementy ciągu podaną wartością.

fill_n()

```
fill_n( iterator początek, n, wartość )
```

Działanie

zastępuje *n* pierwszychh elementów ciągu podaną wartością.

generate()

```
generate( iterator początek, iterator koniec, funkcja )
```

Działanie

zastępuje elementy ciągu wartościami będącymi wynikiem funkcji.

generate_n()

```
generate_n( iterator początek, n, funkcja )
```

Działanie

zastępuje *n* elementów ciągu wartościami będącymi wynikiem funkcji

transform()

```
transform( iterator początek, iterator koniec, iterator nowy_początek, funkcja )
transform( iterator początek, iterator koniec, iterator początek_drugiego, iterator nowy_początek, funkcja_dwuargumentowa )
```

Działanie

wykonuje podaną funkcję dla argumentów ze zbioru (początek, koniec) i zapisuje wyniki do zbioru zaczynającego się w *nowy_początek*. Druga wersja wykonuje funkcję dla pary argumentów, korzystając z drugiego zbioru (wskazywanego przez *początek_drugiego*).

Przykład

program wykonuje funkcję *sqrt* dla każdego z 5 elementów ciągu, zapisując wyniki w nowym ciągu. Niezbędne było określenie którego przeładowania funkcji *sqrt* uzywamy.

```
int main()
{
   double tablica[5] = {2, 3, 9, 16, 25};
   vector<double> v(tablica, tablica+5);
   vector<double> wyniki(5);

   transform(v.begin(), v.end(), wyniki.begin(), (double (*) (double))
sqrt );
   for( int i=0; i<5; i++ )
      cout << wyniki[i] << '\n';
}
```

remove()

Działanie

usuwa elementy o podanej wartości

remove_if()

Działanie

usuwa elementy spełniające warunek

remove_copy()

Działanie

kopiuje ciąg, usuwając elementy o podanej wartości

remove_copy_if()

Działanie

kopiuje ciąg, usuwając elementy spełniające warunek

replace()

Działanie

zastępuje elementy o danej wartości inną wartością

replace_if()

Działanie

zastępuje elementy spełniające warunek

replace_copy()

Działanie

kopiuje ciąg, zastępując elementy o danej wartości inną wartością

replace_copy_if()

Działanie

kopiuje ciąg, zastępując elementy spełniające warunek

Operacje zmieniające kolejność

partition()

```
partition( iterator początek, iterator koniec, funkcja )
```

Działanie

umieszcza elementy spełniające warunek przed tymi które go nie spełniają

stable_partition()

```
stable_partition( iterator początek, iterator koniec, funkcja )
```

Działanie

umieszcza elementy spełniające warunek przed tymi które go nie spełniają, zachowuje wzajemną kolejność

random_shuffle()

```
random_shuffle( iterator początek, iterator koniec )
random_shuffle( iterator początek, iterator koniec, generator_liczb_pseudolosowych )
```

Działanie

w losowy sposób zmienia kolejność elementów ciągu

reverse()

```
reverse( iterator początek, iterator koniec )
```

Działanie

odwraca kolejność elementów w ciągu

reverse_copy()

```
reverse_copy( iterator początek, iterator koniec, iterator początek_kopia )
```

Działanie

kopiuje ciąg do drugiego ciągu, odwracając kolejność elementów

rotate()

```
rotate( iterator początek, iterator nowy_początek, iterator koniec )
```

Działanie

przesuwac elementy w taki sposób aby pierwszym elementem był *nowy_początek*; element go poprzedzający staje się ostatnim

rotate_copy()

```
rotate_copy( iterator początek, iterator nowy_początek, iterator koniec, iterator początek_kopia )
```

Działanie

kopiuje ciąg do drugiego ciągu, przesuwając elementy w taki sposób aby pierwszym elementem był *nowy_początek*; element go poprzedzający staje się ostatnim

unique()

Działanie

usuwa powtórzenia, w taki sposób że wśród sąsiadujących elementów nie ma dwóch takich samych

unique_copy()

```
iterator unique_copy( iterator początek, iterator koniec, iterator początek_kopia )
iterator unique_copy( iterator początek, iterator koniec, iterator początek_kopia, funkcja )
```

Działanie

kopiuje ciąg do drugiego ciągu, w taki sposób że wśród sąsiadujących elementów nie ma dwóch takich samych

swap()

```
swap( element1, element2 )
```

Działanie

zamienia ze sobą dwa elementy.

swap_ranges()

```
swap_ranges( iterator początek, iterator koniec, iterator początek_drugiego )
```

Działanie

zamienia ze sobą dwa zbiory elementów: zbiór (początek, koniec) z drugim (o podanym początku).

iter_swap()

```
iter_swap( iterator element1, iterator element2 )
```

Działanie

zamienia ze sobą dwa elementy wskazywane przez iteratory.

Operacje sortujące

sort()

void sort(RandomAccessIterator start, RandomAccessIterator end) void sort(RandomAccessIterator start, RandomAccessIterator end, Compare cmp)

Działanie

sortuje ciąg rosnąco

Jak używać:

```
#include<algorithm>
...
sort( iterator start, iterator koniec );

//albo

sort( iterator start, iterator koniec, cmp ); //cmp - funkcja porównująca
...
```

W pierwszym przypadku algorytm sort() ustawia elementy w zakresie [start,koniec) w porządku niemalejącym. Gdy wywołujemy sortowanie z trzema parametrami to sortowanie odbywa się względem funkcji porównującej, którą definiujemy.

Przykładowe kody źródłowe

```
vector<int> v;
 v.push_back( 23 );
 v.push_back( -1 );
 v.push_back( 9999 );
 v.push_back( 0 );
 v.push_back( 4 );

 cout << "Przed sortowaniem: ";
 for( int i = 0; i < v.size(); ++i ) {
   cout << v[i] << " ";
 }
```

```
 cout << endl;

 sort( v.begin(), v.end() );

 cout << "Po sortowaniu: ";
 for( int i = 0; i < v.size(); ++i ) {
   cout << v[i] << " ";
 }
 cout << endl;
```

Efektem działania tego programu będzie:

```
Przed sortowaniem: 23 -1 9999 0 4
Po sortowaniu: -1 0 4 23 9999
```

Inny przykład, tym razem z funkcją definiowaną przez programistę:

```
bool cmp( int a, int b ) {
   return a > b;
 }

 ...

 vector<int> v;
 for( int i = 0; i < 10; ++i ) {
   v.push_back(i);
 }

 cout << "Przed: ";
 for( int i = 0; i < 10; ++i ) {
   cout << v[i] << " ";
 }
 cout << endl;

 sort( v.begin(), v.end(), cmp );

 cout << "Po: ";
 for( int i = 0; i < 10; ++i ) {
   cout << v[i] << " ";
 }
 cout << endl;
```

Wyniki działania takiego programu będą następujące:

```
Przed: 0 1 2 3 4 5 6 7 8 9
Po: 9 8 7 6 5 4 3 2 1 0
```

partial_sort()

```
void partial_sort( random_access_iterator start, random_access_iterator middle, random_access_iterator end )
void partial_sort( random_access_iterator start, random_access_iterator middle, random_access_iterator end, StrictWeakOrdering cmp )
```

Działanie

sortuje pierwsze N najmniejszych elementów w ciągu

partial_sort_copy()

```
random_access_iterator partial_sort_copy( input_iterator start,
input_iterator end, random_access_iterator result_start,
random_access_iterator result_end )
random_access_iterator partial_sort_copy( input_iterator start,
input_iterator end, random_access_iterator result_start,
random_access_iterator result_end, StrictWeakOrdering cmp )
```

Działanie

tworzy kopię N najmniejszych elementów ciągu

stable_sort()

```
void stable_sort( random_access_iterator start, random_access_iterator end )
void stable_sort( random_access_iterator start, random_access_iterator end, StrictWeakOrdering cmp )
```

Działanie

sortuje ciąg zachowując wzajemną kolejność dla równych elementów

nth_element()

```
void nth_element( random_access_iterator start, random_access_iterator nth, random_access_iterator end )
void nth_element( random_access_iterator start, random_access_iterator nth, random_access_iterator end, StrictWeakOrdering cmp )
```

Działanie

ciąg jest podzielony na nieposortowane grupy elementów mniejszych i większych od wybranego elementu Nth (odpowiednio po lewej i prawej jego stronie)

Operacje wyszukiwania binarnego

lower_bound()

```
lower_bound( iterator początek, iterator koniec, wartość )
lower_bound( iterator początek, iterator koniec, wartość, funkcja_porównująca )
```

Działanie

zwraca iterator do pierwszego elementu równego lub większego od *wartość*. Jest to pierwsze miejsce, w które można wstawić *wartość* aby zachować uporządkowanie ciągu.

upper_bound()

```
upper_bound( iterator początek, iterator koniec, wartość )
upper_bound( iterator początek, iterator koniec, wartość, funkcja_porównująca )
```

Działanie

zwraca iterator do pierwszego elementu większego od *wartość*. Jest to ostatnie miejsce, w które można wstawić *wartość* aby zachować uporządkowanie ciągu.

binary_search()

```
bool binary_search( iterator początek, iterator koniec, wartość )
bool binary_search( iterator początek, iterator koniec, wartość, funkcja_porównująca )
```

Działanie

zwraca prawdę jeśli *wartość* znajduje się w ciągu (działa w czasie logarytmicznym).

equal_range()

```
pair<> equal_range( iterator początek, iterator koniec, wartość )
pair<> equal_range( iterator początek, iterator koniec, wartość, funkcja_porównująca )
```

Działanie

zwraca parę określającą przedział wewnątrz którego występuje dana wartość (lub ich ciąg), para złożoną z wartości odpowiednio lower_bound i upper_bound.

Operacje na zbiorze

merge()

```
merge( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik_początek )

merge( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik_początek, funkcja_porównująca )
```

Działanie

łączy dwa posortowane ciągi w nowy, posortowany ciąg.

inplace_merge()

```
inplace_merge( iterator początek, iterator środek, iterator koniec )
inplace_merge( iterator początek, iterator środek, iterator koniec, funkcja_porównująca )
```

Działanie

łączy dwie posortowane części ciągu, rozdzielone elementem *środek*, tak że cały ciąg staje się posortowany.

includes()

```
includes( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego )

includes( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, funkcja_porównująca )
```

Działanie

zwraca prawdę jeśli pierwszy ciąg jest podciągiem drugiego.

set_difference()

```
set_difference( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik )

set_difference( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik, funkcja_porównująca )
```

Działanie

tworzy różnicę zbiorów - posortowany ciąg elementów pierwszego ciągu, które nie występują w drugim.

set_intersection()

```
set_intersection( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik )

set_intersection( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik, funkcja_porównująca )
```

Działanie

tworzy przecięcie dwóch zbiorów (zbiór złożony z elementów występujących w obu zbiorach).

set_symmetric_difference()

```
set_symmetric_difference( iterator początek, iterator koniec, iterator początek_2, iterator koniec_2, iterator wynik )

set_symmetric_difference( iterator początek, iterator koniec, iterator początek_2, iterator koniec_2, iterator wynik, funkcja_porównująca )
```

Działanie

tworzy zbiór złożony z elementów występujących w tylko jednym z dwóch ciągów.

set_union()

```
set_union( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik )

set_union( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, iterator wynik, funkcja_porównująca )
```

Działanie

tworzy sumę zbiorów (posortowany zbiór elementów z obu zbiorów, bez powtórzeń).

Operacje na kopcu

is_heap()

```
bool is_heap( iterator początek, iterator koniec )
bool is_heap( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

zwraca prawdę jeśli ciąg tworzy kopiec.

make_heap()

```
make_heap( iterator początek, iterator koniec )
make_heap( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

przekształca ciąg elementów tak aby tworzyły kopiec.

push_heap()

```
push_heap( iterator początek, iterator koniec )
push_heap( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

ostatni element w ciągu zostaje dołączony do struktury kopca.

pop_heap()

```
pop_heap( iterator początek, iterator koniec )
pop_heap( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

usuwa element ze szczytu kopca (o największej wartości), zostaje on przenoszony poza nową strukturę kopca (na koniec ciągu).

sort_heap()

```
sort_heap( iterator początek, iterator koniec )
sort_heap( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

przekształca ciąg o strukturze kopca w ciąg posortowany

Operacje min max

max()

```
wartość max( element1, element2 )
wartość max( element1, element2, funkcja_porównująca )
```

Działanie

zwraca większy z dwóch elementów

max_element()

```
iterator max_element( iterator początek, iterator koniec )
iterator max_element( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

zwraca największy z elementów w ciągu

min()

```
wartość min( element1, element2 )
wartość min( element1, element2, funkcja_porównująca )
```

Działanie

zwraca mniejszy z dwóch elementów

min_element()

```
iterator min_element( iterator początek, iterator koniec )
iterator min_element( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

zwraca najmniejszy z elementów w ciągu

lexicographical_compare()

```
bool lexicographical_compare( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego )
bool lexicographical_compare( iterator początek, iterator koniec, iterator początek_drugiego, iterator koniec_drugiego, funkcja )
```

Działanie

sprawdza czy jeden ciąg poprzedza leksykograficznie drugi ciąg, zwraca prawdę jeśli poprzedza.

next_permutation()

```
bool next_permutation( iterator początek, iterator koniec )
bool next_permutation( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

przekształca ciąg elementów w leksykograficznie następną permutację. Zwraca prawdę przy powodzeniu.

prev_permutation()

```
bool prev_permutation( iterator początek, iterator koniec )
bool prev_permutation( iterator początek, iterator koniec, funkcja_porównująca )
```

Działanie

przekształca ciąg elementów w leksykograficznie poprzedzającą permutację. Zwraca prawdę przy powodzeniu.

Operacje numeryczne

accumulate()

```
accumulate( iterator początek, iterator koniec, wartość )
accumulate( iterator początek, iterator koniec, wartość, funkcja )
```

Działanie

sumuje ciąg elementów

inner_product()

```
inner_product( iterator początek, iterator koniec, iterator początek_wyniku, wartość )
inner_product( iterator początek, iterator koniec, iterator początek_wyniku, wartość, funkcja_dodawania, funkcja_mnozenia )
```

Działanie

oblicza iloczyn skalarny na elementach dwóch ciągów

adjacent_difference()

```
adjacent_difference( iterator początek, iterator koniec, iterator początek_wyniku)
adjacent_difference( iterator początek, iterator koniec, iterator początek_wyniku, funkcja )
```

Działanie

oblicza różnice pomiędzy sąsiadującymi elementami w ciągu

partial_sum()

```
partial_sum( iterator początek, iterator koniec, iterator początek_wyniku)
partial_sum( iterator początek, iterator koniec, iterator początek_wyniku, funkcja )
```

Działanie

oblicza sumy częściowe ciągu elementów

Źródła i autorzy artykułu

Algorytmy w STL *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=168879 *Autorzy*: Derbeth, Karol Dąbrowski, Kszyh, Lethern, PDD, 4 anonimowych edycji

Operacje niemodyfikujące *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=170825 *Autorzy*: Lethern, 3 anonimowych edycji

Operacje niemodyfikujące - szukanie *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158004 *Autorzy*: Lethern

Operacje modyfikujące *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158058 *Autorzy*: Lethern

Operacje zmieniające kolejność *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158060 *Autorzy*: Lethern

Operacje sortujące *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158040 *Autorzy*: Lethern

Operacje wyszukiwania binarnego *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158069 *Autorzy*: Lethern

Operacje na zbiorze *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158067 *Autorzy*: Lethern

Operacje na kopcu *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158066 *Autorzy*: Lethern

Operacje min max *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158064 *Autorzy*: Lethern

Operacje numeryczne *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=158063 *Autorzy*: Lethern

Licencja