STL: algorytmy modyfikujące

```
template<class ForwardIterator, class Type>
void replace( ForwardIterator _First, ForwardIterator _Last,
              const Type& _OldVal, const Type& _NewVal );
```

- w sekwencji określanej przez parę iteratorów [_First,_Last) zamienia wszystkie wystąpienia wartości _OldVal wartością _NewVal.

48

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::replace
#include <vector>       // std::vector

int main () {
  int myT[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
  std::vector<int> V(myT, myT+8);   // 10 20 30 30 20 10 10 20

  std::replace (V.begin(), V.end(), 20, 99); // 10 99 30 30 99 10 10 99

  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


49

STL: algorytmy modyfikujące

```
template<class ForwardIterator, class Predicate, class Type>
void replace_if( ForwardIterator _First,
                 ForwardIterator _Last,
                 Predicate _Pred, const Type& _Val );
```

- w sekwencji określanej przez parę iteratorów [_First,_Last) zamienia wszystkie elementy spełniające predykat _Pred wartością _NewVal.

50

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::replace_if
#include <vector>       // std::vector

bool Nieparzysty (int i) { return ((i%2)==1); }

int main () {
  std::vector<int> V;
  for (int i=1; i<10; i++) V.push_back(i);              // 1 2 3 4 5 6 7 8 9

  std::replace_if (V.begin(), V.end(), Nieparzysty, 0);  // 0 2 0 4 0 6 0 8 0

  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


51

STL: algorytmy modyfikujące

```
template<class InputIterator, class OutputIterator,
                                              class Type>
OutputIterator replace_copy(
              InputIterator _First, InputIterator _Last,
              OutputIterator _Result, const Type& _OldVal,
              const Type& _NewVal );
```

- z sekwencji określanej przez parę iteratorów [_First,_Last) kopiuje do sekwencji, której pierwszy element jest wskazywany przez _Result, wszystkie elementy, ale po drodze zamienia elementy o wartości _OldVal elementami o wartości _NewVal

52

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::replace_copy
#include <vector>       // std::vector

int main () {
  int myT[] = { 10, 20, 30, 30, 20, 10, 10, 20 };

  std::vector<int> V(8);
  std::replace_copy (myT, myT+8, V.begin(), 20, 99);

  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


53

STL: algorytmy modyfikujące

```
template<class InputIterator, class OutputIterator, class
   Predicate, class Type>
OutputIterator replace_copy_if(
 InputIterator _First, InputIterator _Last,
 OutputIterator _Result, Predicate _Pred, const Type& _Val );
```

- z sekwencji określanej przez parę iteratorów [_First,_Last) kopiuje do sekwencji, której pierwszy element jest wskazywany przez _Result, wszystkie elementy, ale po drodze zamienia elementy spełniające predykat _Pred wartością _NewVal.

54

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::replace_copy_if
#include <vector>       // std::vector

bool Nieparzysty (int i) { return ((i%2)==1); }
int main () {
  std::vector<int> V, W;

  for (int i=1; i<10; i++) V.push_back(i);          // 1 2 3 4 5 6 7 8 9
  W.resize(V.size());   // allocate space
  std::replace_copy_if (V.begin(), V.end(), W.begin(), Nieparzysty, 0);
                                                    // 0 2 0 4 0 6 0 8 0
  std::cout << "W zawiera:";
  for (std::vector<int>::iterator it=W.begin(); it!=W.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


55

STL: algorytmy modyfikujące

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(
   InputIterator _First, InputIterator _Last,
   OutputIterator _Result );
```

- z sekwencji określanej przez parę iteratorów [_First,_Last) kopiuje do sekwencji, której pierwszy element jest wskazywany przez _Result, wartości bez powtórzeń, tj. tak, aby każda wartość w sekwencji wynikowej była unikalna (występowała w niej tylko raz),
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim argumentem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

56

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::unique_copy, std::sort, std::distance
#include <vector>       // std::vector

int main () {
  int myT[] = {10,20,20,20,30,30,20,20,10};
  std::vector<int> V (9);                   // V: 0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;
  it=std::unique_copy (myT,myT+9,V.begin()); // V: 10 20 30 20 10 0  0  0  0
                                            //             ^
  std::cout << "V zawiera:";
  for (it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


57

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::unique_copy, std::sort, std::distance
#include <vector>       // std::vector
bool myfunction (int i, int j) {  return (i==j); };
int main () {
  int myT[] = {10,20,20,20,30,30,20,20,10};
  std::vector<int> V (9);                   // V: 0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;
  it=std::unique_copy (myT,myT+9,V.begin(), myfunction);
                                            // V: 10 20 30 20 10 0  0  0  0
                                            //             ^
  std::cout << "V zawiera:";
  for (it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


58

STL: algorytmy modyfikujące

```
template<class BidirectionalIterator>
void reverse( BidirectionalIterator _First,
              BidirectionalIterator _Last );
```

- odwraca kolejność elementów w sekwencji określanej przez parę iteratorów [_First,_Last) .

59

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::reverse
#include <vector>       // std::vector

int main () {
  std::vector<int> V;
  for (int i=1; i<10; ++i) V.push_back(i); // 1 2 3 4 5 6 7 8 9

  std::reverse(V.begin(),V.end());         // 9 8 7 6 5 4 3 2 1

  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


STL: algorytmy modyfikujące

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(
        BidirectionalIterator _First,
        BidirectionalIterator _Last,
        OutputIterator _Result );
```

- z sekwencji określanej przez parę iteratorów `[_First,_Last)` kopiuje do sekwencji, której pierwszy element jest wskazywany przez `_Result`, wszystkie elementy ale w odwrotnej kolejności.

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::reverse_copy
#include <vector>       // std::vector

int main () {
  int myT[] ={1,2,3,4,5,6,7,8,9};
  std::vector<int> V;

  V.resize(9);    // allocate space
  std::reverse_copy (myT, myT+9, V.begin());

  std::cout << "V contains:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


STL: algorytmy modyfikujące

```
template<class ForwardIterator>
void rotate( ForwardIterator _First,
             ForwardIterator _Middle,
             ForwardIterator _Last );
```

- „obraca” elementy w sekwencji określanej przez parę iteratorów `[_First,_Last)` zamieniając lewą i prawą połowę elementów sekwencji miejscami, przy czym iterator `_Middle` wskazuje środek.

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::rotate
#include <vector>       // std::vector

int main () {
  std::vector<int> V;
  for (int i=1; i<10; ++i) V.push_back(i); // 1 2 3 4 5 6 7 8 9

  std::rotate(V.begin(),V.begin()+3,V.end());
                                           // 4 5 6 7 8 9 1 2 3
  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


STL: algorytmy modyfikujące

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy( ForwardIterator _First,
   ForwardIterator _Middle, ForwardIterator _Last,
   OutputIterator _Result );
```

- do sekwencji wskazywanej przez `_Result` kopiuje elementy z sekwencji określanej przez parę iteratorów `[_First,_Last)` w innym porządku, tj. zamieniając lewą i prawą połowę elementów sekwencji miejscami, przy czym iterator `_Middle` wskazuje środek.

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::rotate_copy
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,40,50,60,70};
  std::vector<int> V (7);

  std::rotate_copy(myints,myints+3,myints+7,V.begin());

  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


STL: algorytmy modyfikujące

```
template <class RandomAccessIterator, class
   RandomNumberGenerator>
void random_shuffle ( RandomAccessIterator first,
                      RandomAccessIterator last);

template <class RandomAccessIterator, class
   RandomNumberGenerator>
void random_shuffle ( RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomNumberGenerator& rand )
```

miesza kolejność elementów w sekwencji.

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::random_shuffle
#include <vector>       // std::vector
#include <ctime>        // std::time
#include <cstdlib>      // std::rand, std::srand
int myrandom (int i) { return std::rand()%i;}
int main () {
  std::srand ( unsigned ( std::time(0) ) );
  std::vector<int> V;
  for (int i=1; i<10; ++i) V.push_back(i); // 1 2 3 4 5 6 7 8 9
  std::random_shuffle ( V.begin(), V.end() );
  std::random_shuffle ( V.begin(), V.end(), myrandom);
  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


STL: algorytmy modyfikujące

Jak miesza **`random_shuffle`**

```
template <class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle (   RandomAccessIterator first,
                          RandomAccessIterator last,
                          RandomNumberGenerator& gen)
{
  iterator_traits<RandomAccessIterator>::difference_type i, n;
  n = (last-first);
  for (i=n-1; i>0; --i) {
    swap (first[i],first[gen(i+1)]);
  }
}
```

Wymaga kontenera posiadającego iteratory dostępu swobodnego (**`vector`**,**`deque`**).
Dlatego nie można stosować np. do **`list`**.

STL: algorytmy modyfikujące

```
template < class InputIterator, class OutputIterator,
                                 class UnaryOperator >
  OutputIterator transform (
       InputIterator first1, InputIterator last1,
       OutputIterator result, UnaryOperator op )
```

- modyfikuje elementy z sekwencji operatorem jednoargumentowym i wynik modyfikacji zapisuje do innej sekwencji.

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::transform
#include <vector>       // std::vector
int op_increase (int i) { return ++i; }
int main () {
  std::vector<int> V;
  std::vector<int> W;
for (int i=1; i<6; i++)
     V.push_back (i*10);                          // V: 10 20 30 40 50
  W.resize(V.size());       // allocate space
  std::transform (V.begin(), V.end(), W.begin(), op_increase);
                                                  // W: 11 21 31 41 51
  std::cout << „W zawiera:";
  for (std::vector<int>::iterator it=W.begin(); it!=W.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

STL: algorytmy modyfikujące

```
template<class InputIterator1, class InputIterator2, class
   OutputIterator, class BinaryFunction>
OutputIterator transform(
              InputIterator1 _First1, InputIterator1 _Last1,
              InputIterator2 _First2, OutputIterator _Result,
              BinaryFunction _Func );
```

- transformuje elementy pochodzące z dwóch sekwencji operatorem dwuargumentowym, pierwsza sekwencja jest wskazywana przez iteratory `[_First1, _Last1)`, pierwszy element drugiej jest wskazywany przez iterator `_First2`, wynik zapisywany jest do sekwencji, której pierwszy element jest wskazywany przez `_Result`, natomiast obiekt funkcyjny jest wskazywany przez `_Func`.

STL: algorytmy modyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::transform
#include <vector>       // std::vector
#include <functional>   // std::plus
int op_increase (int i) { return ++i; }
int main () {
  std::vector<int> V;
  std::vector<int> W;
for (int i=1; i<6; i++)
    V.push_back (i*10);                          // V: 10 20 30 40 50
  W.resize(V.size());       // allocate space
  std::transform (V.begin(), V.end(), W.begin(), op_increase);
                                                 // W: 11 21 31 41 51
  // std::plus adds together its two arguments:
  std::transform (V.begin(), V.end(), W.begin(), V.begin(), std::plus<int>());
                                                 // V: 21 41 61 81 101
  return 0;
}
```


STL: algorytmy modyfikujące

Podsumowanie:

1. fill,
2. fill_n,
3. generate,
4. generate_n,
5. copy,
6. copy_backward,
7. swap,
8. swap_ranges,
9. replace,
10. replace_if,
11. replace_copy,
12. replace_copy_if,
13. unique_copy,
14. reverse_copy,
15. rotate,
16. rotate_copy,
17. random_shuffle,
18. transform

STL: algorytmy sortujące

1. partition
2. stable_partition
3. sort
4. stable_sort
5. partial_sort
6. partial_sort_copy
7. nth_element
8. lower_bound
9. upper_bound
10. equal_range
11. binary_search
12. merge
13. inplace_merge
14. includes
15. set_union
16. set_intersection
17. set_difference
18. set_symmetric_difference
19. lexicographical_compare
20. next_permutation

STL: algorytmy sortujące

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator partition(
       BidirectionalIterator _First,
       BidirectionalIterator _Last,
       BinaryPredicate _Comp );
```

- porządkuje na nowo sekwencję elementów tak, że w pierwszej kolejności w sekwencji znajdują się elementy spełniające warunek podany w predykacie, a następnie pozostałe, niespełniające warunku. Zwracany iterator wskazuje element graniczny, tj. pierwszy element niespełniający warunku z predykatu.

STL: algorytmy sortujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition
#include <vector>       // std::vector
bool Nieparzyste (int i) { return (i%2)==1; }
int main () {
  std::vector<int> V;
  for (int i=1; i<10; ++i) V.push_back(i); // 1 2 3 4 5 6 7 8 9
  std::vector<int>::iterator bound;
  bound = std::partition (V.begin(), V.end(), Nieparzyste);
 std::cout << "nieparzyste elementy:";
  for (std::vector<int>::iterator it=V.begin(); it!=bound; ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  std::cout << "parzyste elementy:";
  for (std::vector<int>::iterator it=bound; it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;}
```

STL: algorytmy sortujące

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(
        BidirectionalIterator _First,
        BidirectionalIterator _Last,
        Predicate _Pred );
```

- porządkuje na nowo sekwencję elementów tak, że w pierwszej kolejności w sekwencji znajdują się elementy spełniające warunek podany w predykacie, a następnie pozostałe, niespełniające warunku. Zwracany iterator wskazuje element graniczny, tj. pierwszy element niespełniający warunku z predykatu
- porządkowanie jest *stabilne.*

Co to znaczy, że jest stabilne?

STL: algorytmy sortujące

- Szereg algorytmów porządkujących występuje w wersjach stabilnych i niestabilnych.
- Wersje stabilne zachowują wzajemne uporządkowanie elementów traktowanych jako identyczne z punktu widzenia funkcji porządkującej, np. sekwencja: C_1,B_1,C_2,A_1,B_2,A_2 po posortowaniu pod względem dużej litery algorytmem stabilnym będzie miała postać: A_1,A_2,B_1,B_2,C_1,C_2 natomiast po posortowaniu algorytmem niestabilnym może mieć np. postać: A_2,A_1,B_1,B_2,C_2,C_1

Np. implementacja algorytmu 'sort' w STL jest oparta na quicksort, a więc jest niestabilna. Istnieje jednak również wersja 'stable_sort', która zachowuje wzajemne uporządkowanie elementów identycznych.

STL: algorytmy sortujące

```
template<class RandomAccessIterator>
void sort(    RandomAccessIterator _First,
              RandomAccessIterator _Last );

template<class BidirectionalIterator>
void stable_sort(   BidirectionalIterator _First,
                    BidirectionalIterator _Last );
```

- porządkuje elementy w sekwencji określanej przez parę iteratorów [*_First,_Last*) w porządku nierosnącym,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

STL: algorytmy sortujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::stable_sort
#include <vector>       // std::vector

bool porownajInt (double i,double j) { return (int(i)<int(j)); };

int main () {
  double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
  std::vector<double> V;
  V.assign(mydoubles,mydoubles+8);
  std::cout << "sortowanie porownajInt:";
  std::stable_sort (V.begin(), V.end(), porownajInt);
  for (std::vector<double>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


STL: algorytmy sortujące

```
template<class RandomAccessIterator>
void partial_sort(
   RandomAccessIterator _First,
   RandomAccessIterator _SortEnd,
   RandomAccessIterator _Last );
```

- sortuje ograniczoną liczbę elementów z sekwencji określanej przez parę iteratorów [*_First,_Last*) w porządku nierosnącym, którą da się zmieścić w zakresie [*_First,_SortEnd*). Reszta elementów trafia do [*_SortEnd,_Last*) w nieokreślonej kolejności,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

STL: algorytmy sortujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partial_sort
#include <vector>       // std::vector

bool porownaj (int i,int j) { return (i<j); }

int main () {
  int myints[] = {9,8,7,6,5,4,3,2,1};
  std::vector<int> V (myints, myints+9);
  std::partial_sort (V.begin(), V.begin()+5, V.end(), porownaj);
  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

STL: algorytmy sortujące

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
   InputIterator _First1, InputIterator _Last1,
   RandomAccessIterator _First2, RandomAccessIterator _Last2
   );
```

- Sortuje liczbę elementów z sekwencji określanej przez parę iteratorów [*_First1,_Last1*) w porządku nierosnącym, którą da się umieścić w zakresie [*_First2,_Last2*) i kopiuje posortowane elementy do sekwencji [*_First2,_Last2*),
- Jeżeli [*_First1,_Last1*) jest mniejszy od [*_First2,_Last2*), kopiowana jest mniejsza liczba elementów,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

STL: algorytmy sortujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partial_sort_copy
#include <vector>       // std::vector

bool porownaj (int i,int j) { return (i<j); };

int main () {
  int myints[] = {9,8,7,6,5,4,3,2,1};
  std::vector<int> V (5);
  std::partial_sort_copy (myints, myints+9, V.begin(), V.end(), porownaj);
  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


STL: algorytmy sortujące

```
template<class RandomAccessIterator>
void nth_element(
   RandomAccessIterator _First,
   RandomAccessIterator _Nth,
   RandomAccessIterator _Last );
```

- *częściowo* sortuje sekwencję określaną przez parę iteratorów [*_First,_Last*) w porządku nierosnącym. To znaczy, że:
  po posortowaniu elementy mniejsze od elementu wskazywane przez *_Nth* umieszczane są w lewej części sekwencji, tj. [*_First, _Nth*), natomiast większe w prawej – [*_Nth,_Last*). Jednak żadna z podsekwencji nie musi być posortowana,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::nth_element, std::random_shuffle
#include <vector>       // std::vector

bool porownaj (int i,int j) { return (i>j); }

int main () {
  std::vector<int> V;
  for (int i=1; i<10; i++) V.push_back(i);   // 1 2 3 4 5 6 7 8 9
  std::random_shuffle (V.begin(), V.end());
  std::nth_element (V.begin(), V.begin()+5, V.end(), porownaj);
  std::cout << "V zawiera:";
  for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class ForwardIterator, class Type>
ForwardIterator lower_bound(
      ForwardIterator _First, ForwardIterator _Last,
      const Type& _Val );
```

- zwraca iterator na pierwsze wystąpienie w posortowanej sekwencji [*_First,_Last*) elementu o wartości *_Val*,
- jeżeli w sekwencji brak takiego elementu, zwracany jest iterator do miejsca, w którym element ten powinien się znajdować, tj. do pierwszego elementu większego od poszukiwanej wartości,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class ForwardIterator, class Type>
ForwardIterator upper_bound(
      ForwardIterator _First, ForwardIterator _Last,
      const Type& _Val );
```

- zwraca iterator na pierwszy element za ostatnim elementem o wartości *_Val* w posortowanej sekwencji [*_First,_Last*),
- jeżeli w sekwencji brak takiego elementu, zwracany jest iterator do miejsca, w którym element ten powinien się znajdować, tj. do pierwszego elementu większego od poszukiwanej wartości,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound, std::upper_bound, std::sort
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20

  std::sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

  std::vector<int>::iterator low,up;
  low=std::lower_bound (v.begin(), v.end(), 20); //          ^
  up= std::upper_bound (v.begin(), v.end(), 20); //                   ^

  std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
  std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
  return 0;
}
```


90

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class ForwardIterator, class Type>
pair<ForwardIterator, ForwardIterator> equal_range(
      ForwardIterator _First, ForwardIterator _Last,
      const Type& _Val );
```

- zwraca parę iteratorów ograniczających w sekwencji [*_First,_Last*) podsekwencję elementów o wartości *_Val*,
- jeżeli w sekwencji brak takiego elementu, iteratory wskazują na miejsce, w którym element ten powinien się znajdować, tj. do pierwszego elementu większego od poszukiwanej wartości,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań,
- Uwaga: elementy powinny być posortowane względem operatora < (*elementy są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`) lub podanego obiektu funkcyjnego/funkcji służącej do porównań.

91

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::equal_range, std::sort
#include <vector>       // std::vector
bool mygreater (int i,int j) { return (i>j); }
int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> V(myints,myints+8);             // 10 20 30 30 20 10 10 20
  std::pair<std::vector<int>::iterator,std::vector<int>::iterator> bounds;
  std::sort (v.begin(), v.end(), mygreater);       // 30 30 20 20 20 10 10 10
  bounds=std::equal_range (
        v.begin(), v.end(), 20, mygreater);        //       ^        ^
  std::cout << "bounds at positions " << (bounds.first - v.begin());
  std::cout << " and " << (bounds.second - v.begin()) << '\n';
  return 0;
}
```


92

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template <class ForwardIterator, class T>
bool binary_search ( ForwardIterator _First,
                     ForwardIterator _Last,
                     const T& _Val);
```

- zwraca wartość logiczną **true**, jeżeli w posortowanej sekwencji [*_First,_Last*) znajduje się element o wartości *_Val (dwa elementy* **a** *i* **b** *są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`),
- optymalizuje liczbę porównań poprzez dokonywanie porównań między niesąsiadującymi elementami z posortowanego zakresu,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

93

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2,
                                class OutputIterator>
OutputIterator merge(
      InputIterator1 _First1, InputIterator1 _Last1,
      InputIterator2 _First2, InputIterator2 _Last2,
            OutputIterator _Result );
```

- łączy dwa identycznie posortowane zakresy elementów [*_First1,_Last1*) i [*_First2,_Last2*) w jeden posortowany zakres docelowy, którego początek jest wskazywany przez *_Result,*
- Wymagany jest kontener docelowy inny niż źródłowy, tj. nie może skierować wyniku swojego działania do kontenera, skąd pochodziły dane wejściowej
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

94

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::merge, std::sort
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};          // pierwszy kontener
  int second[] = {50,40,30,20,10};        // drugi kontener
  std::vector<int> v(10);                 // wynikowy kontener
  std::sort (first,first+5);
  std::sort (second,second+5);
  std::merge (first,first+5,second,second+5,v.begin());
  std::cout << "Wynikowy wektor zawiera:"; // 5 10 10 15 20 20 25 30 40 50
  for (std::vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```


95

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class BidirectionalIterator>
void inplace_merge( BidirectionalIterator _First,
                    BidirectionalIterator _Middle,
                    BidirectionalIterator _Last );
```

- łączy dwa posortowane zakresy elementów [*_First*,*_Middle*) i [*_Middle*,*_Last*) w jeden sumaryczny zakres docelowy [*_First*,*_Last*). Próbuje alokować pomocniczy bufor na dane, aby zwiększyć efektywność, a jeżeli jest to niemożliwe, używa mniej efektywnego algorytmu.
- Różnice **merge** vs. **inplace_merge**:
  - **merge**: liczba porównań – co najwyżej `distance(first1, last1) + distance(first2, last2) - 1`
  - **inplace_merge**: dokładnie **N-1** porównań, jeżeli dostępny jest wystarczający obszar pamięci, w przeciwnym razie: **N ·log(N)** gdzie **N = distance(first, last)**.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

Różnice **merge** vs. **inplace_merge**:

```
std::inplace_merge(a.begin(), a.begin() + n, a.end());
```

nie jest równoważny:

```
std::merge(a.begin(), a.begin() + n,
           a.begin() + n, a.end(), a.begin());
```

ponieważ **merge** nie może skierować wyniku swojego działania do **a** (tj. do tego samego kontenera, z którego pochodziły dane wejściowe).

inplace_merge *przydaje się szczególnie wtedy, gdy działamy na systemie z niewielką pamięcią.*

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::inplace_merge, std::sort, std::copy
#include <vector>       // std::vector
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);
  std::vector<int>::iterator it;
  std::sort (first,first+5);
  std::sort (second,second+5);
  it=std::copy (first, first+5, v.begin());
  std::copy (second,second+5,it);
  std::inplace_merge (v.begin(),v.begin()+5,v.end());
  std::cout << " Wynikowy wektor zawiera :"; // 5 10 10 15 20 20 25 30 40 50
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 _First1, InputIterator1 _Last1,
              InputIterator2 _First2, InputIterator2 _Last2);
```

- sprawdza, czy wszystkie wartości z jednego posortowanego zakresu [*_First1*,*_Last1*) są zawarte w drugim posortowanym zakresie (*_First2*,*_Last2*) (tj. czy pierwszy jest podzbiorem drugiego – *dwa elementy* **a** *i* **b** *są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`),
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::includes, std::sort

bool porownaj (int i, int j) { return i<j; }

int main () {
  int C1[] = {5,10,15,20,25,30,35,40,45,50};
  int C2[] = {40,30,20,10};
  std::sort (C1,C1+10);
  std::sort (C2,C2+4);
  if ( std::includes(C1,C1+10,C2,C2+4, porownaj) )
    std::cout << "C1 zawiera C2!\n";

  return 0;
}
```


STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2, class
  OutputIterator>
OutputIterator set_union(
      InputIterator1 _First1, InputIterator1 _Last1,
      InputIterator2 _First2, InputIterator2 _Last2,
      OutputIterator _Result );
```

- Łączy wszystkie elementy znajdujące się w przynajmniej jednym z dwóch posortowanych zakresów tworząc jeden wspólny posortowany zakres wynikowy (suma zbiorów) – *dwa elementy* **a** *i* **b** *są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`,
- Elementy z drugiego zakresu, które występują w pierwszym, nie są kopiowane
- Elementy w obydwu zakresach muszą być posortowane wg tego samego kryterium. Wynikowy porządek jest taki sam jak dla zakresów wejściowych.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::set_union, std::sort
#include <vector>       // std::vector
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                          // 0  0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;
  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50
  it=std::set_union (first, first+5, second, second+5, v.begin());
                                                   // 5 10 15 20 25 30 40 50  0  0
  v.resize(it-v.begin());                          // 5 10 15 20 25 30 40 50
  std::cout << "Suma ma " << (v.size()) << " elementów:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


102

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2, class
  OutputIterator>
OutputIterator set_intersection(
        InputIterator1 _First1, InputIterator1 _Last1,
        InputIterator2 _First2, InputIterator2 _Last2,
        OutputIterator _Result );
```

- łączy wszystkie elementy, które należą do obydwu posortowanych zakresów w jeden wspólny posortowany zakres wynikowy (przecięcie zbiorów) – *dwa elementy* **a** *i* **b** *są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`,
- Wszystkie elementy występujące w obydwu zakresach są kopiowane z pierwszego zakresu w takiej kolejności, w jakiej w nim występują.
- Elementy w obydwu zakresach muszą być posortowane wg tego samego kryterium. Wynikowy porządek jest taki sam jak dla zakresów wejściowych.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

103

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::set_intersection, std::sort
#include <vector>       // std::vector
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                          // 0  0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;
  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50
  it=std::set_intersection (first, first+5, second, second+5, v.begin());
                                                   // 10 20 0  0  0  0  0  0  0  0
  v.resize(it-v.begin());                          // 10 20
  std::cout << "Część wspólna ma " << (v.size()) << " elementów:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


104

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2, class
  OutputIterator>
OutputIterator set_difference(
        InputIterator1 _First1, InputIterator1 _Last1,
        InputIterator2 _First2, InputIterator2 _Last2,
        OutputIterator _Result );
```

- łączy wszystkie elementy, które należą do pierwszego, ale nie do drugiego z zakresów (obydwa muszą być posortowane) w jeden posortowany zakres wynikowy (różnica zbiorów) – *dwa elementy* **a** *i* **b** *są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`,
- Elementy są kopiowane do zakresu wynikowego wyłącznie z pierwszego zakresu w kolejności takiej, jak występują w tym przedziale.
- Elementy w obydwu zakresach muszą być posortowane wg tego samego kryterium. Wynikowy porządek jest taki sam jak dla zakresów wejściowych.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

105

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::set_difference, std::sort
#include <vector>       // std::vector
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                          // 0  0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;
  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50
  it=std::set_difference (first, first+5, second, second+5, v.begin());
                                                   //  5 15 25  0  0  0  0  0  0  0
  v.resize(it-v.begin());                          //  5 15 25
  std::cout << "Różnica ma " << (v.size()) << " elementów:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


106

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2, class
  OutputIterator>
OutputIterator set_symmetric_difference(
        InputIterator1 _First1, InputIterator1 _Last1,
        InputIterator2 _First2, InputIterator2 _Last2,
        OutputIterator _Result );
```

- łączy wszystkie elementy, które należą albo do jednego, albo do drugiego zakresu (obydwa muszą być posortowane) w jeden posortowany zakres wynikowy (różnica symetryczna zbiorów) – *dwa elementy* **a** *i* **b** *są uważane za równe, jeżeli:* `(!(a<b) && !(b<a))`,
- Kopiowane są elementy z obydwu zakresów z zachowaniem kolejności.
- Elementy w obydwu zakresach muszą być posortowane wg tego samego kryterium. Wynikowy porządek jest taki sam jak dla zakresów wejściowych.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

107

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::set_symmetric_difference, std::sort
#include <vector>       // std::vector
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                      // 0  0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;
  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50
  it=std::set_symmetric_difference(first,first+5,second,second+5,v.begin());
                                 //  5 15 25 30 40 50  0  0  0  0
  v.resize(it-v.begin());        //  5 15 25 30 40 50
  std::cout << "Różnica symetr. ma " << (v.size()) << " elementów:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```


108

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
        InputIterator1 _First1, InputIterator1 _Last1,
        InputIterator2 _First2, InputIterator2 _Last2 );
```

- porównuje dwa przedziały element po elemencie aby wskazać ten mniejszy. Zwraca **`true`**, jeżeli pierwszy jest leksykograficznie mniejszy, lub **`false`** – w przeciwnym przypadku,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

109

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout, std::boolalpha
#include <algorithm>    // std::lexicographical_compare
#include <cctype>       // std::tolower
bool porownaj(char c1, char c2)
{ return std::tolower(c1)<std::tolower(c2); };
int main () {
  char V1[]="Apple";
  char V2[]="apartment";
  std::cout << std::boolalpha;
  std::cout << "Porównuje V1 i V2 leksykograficznie (V1<V2):\n";
  std::cout << "Operator domyślny (<): ";
  std::cout << std::lexicographical_compare(V1,V1+5,V2,V2+9);
  std::cout << '\n';
  std::cout << "Operator porownaj: ";
  std::cout << std::lexicographical_compare(V1,V1+5,V2,V2+9,porownaj);
  std::cout << '\n';
  return 0;
}
```


110

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
template <class BidirectionalIterator>
  bool next_permutation (BidirectionalIterator first,
                         BidirectionalIterator last);
```

- zmienia porządek elementów w sekwencji *`[first, last)`* na następną, leksykograficznie większą permutację,
- Za pierwszą permutację przyjmujemy sekwencję posortowaną rosnąco. Za ostatnią – malejąco.
- Zwraca `true`, jeżeli istnieje możliwość dokonania kolejnej permutacji i wykonuje ją. W przeciwnym przypadku, tj. kiedy osiągnięto już ostatnią permutację, zmienia prządek na pierwszą permutację i zwraca `false`.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

111

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream>     // std::cout
#include <algorithm>    // std::next_permutation, std::sort

int main () {
  int myints[] = {1,2,3};

  std::sort (myints,myints+3);     // ustawiamy początkową permutację

  std::cout << "The 3! possible permutations with 3 elements:\n";
  do {
    std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
  } while ( std::next_permutation(myints,myints+3) );

  std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' <<
   myints[2] << '\n';
  return 0;
}
```


112

STL: algorytmy sortujące

Podsumowanie

1. `partition`
2. `stable_partition`
3. `sort`
4. `stable_sort`
5. `partial_sort`
6. `partial_sort_copy`
7. `nth_element`
8. `lower_bound`
9. `upper_bound`
10. `equal_range`
11. `binary_search`
12. `merge`
13. `inplace_merge`
14. `includes`
15. `set_union`
16. `set_intersection`
17. `set_difference`
18. `set_symmetric_difference`
19. `lexicographical_compare`
20. `next_permutation`

113