

STL: algorytmy modyfikujące

```
template< typename ForwardIterator1,
          typename ForwardIterator2>
void iter_swap( ForwardIterator1 _Left,
                ForwardIterator2 _Right );
```

- zamienia miejscami dwie wartości wskazywane przez dwa iteratory.

© UKSW, WMP, SNS, Warszawa

45

45

STL: algorytmy modyfikujące

```
template< typename ForwardIterator1,
          typename ForwardIterator2>
ForwardIterator2 swap_ranges(
    ForwardIterator1 _First1, ForwardIterator1 _Last1,
    ForwardIterator2 _First2 );
```

- zamienia miejscami wartości z dwóch zakresów, jeden wskazywany przez parę iteratorów [*_First*, *_Last*), a drugi zaczynający się od elementu wskazywanego przez iterator *_First2*.

© UKSW, WMP, SNS, Warszawa

46

46

STL: algorytmy modyfikujące

```
#include <algorithm> // std::swap_ranges
#include <vector>    // std::vector
#include <deque>     // std::deque

std::vector<int> v1;
std::deque<int> d1;
for ( int i = 0 ; i <= 5 ; i++ )
    v1.push_back( i );
for ( int ii =4 ; ii <= 9 ; ii++ )
    d1.push_back( 6 );

std::swap_ranges ( v1.begin ( ) , v1.end ( ) , d1.begin ( ) );
```

© UKSW, WMP, SNS, Warszawa

47

47

STL: algorytmy modyfikujące

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

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

© UKSW, WMP, SNS, Warszawa

48

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;
}
```

© UKSW, WMP, SNS, Warszawa

49

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 określonej przez parę iteratorów [*_First*, *_Last*) zamienia wszystkie elementy spełniające predykat *_Pred* wartością *_NewVal*.

© UKSW, WMP, SNS, Warszawa

50

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;
}
```

© UKSW, WMP, SNS, Warszawa

51

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 określonej 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*

© UKSW, WMP, SNS, Warszawa

52

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;
}
```

© UKSW, WMP, SNS, Warszawa

53

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 określonej 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*.

© UKSW, WMP, SNS, Warszawa

54

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;
}
```

© UKSW, WMP, SNS, Warszawa

55

55

STL: algorytmy modyfikujące

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

- z sekwencji określonej 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.

© UKSW, WMP, SNS, Warszawa

56

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;
}
```

© UKSW, WMP, SNS, Warszawa

57

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;
}
```

© UKSW, WMP, SNS, Warszawa

58

58

STL: algorytmy modyfikujące

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

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

© UKSW, WMP, SNS, Warszawa

59

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;
}
```

© UKSW, WMP, SNS, Warszawa

60

60

STL: algorytmy modyfikujące

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

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

© UKSW, WMP, SNS, Warszawa

61

61

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;
}
```

© UKSW, WMP, SNS, Warszawa

62

62

STL: algorytmy modyfikujące

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

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

© UKSW, WMP, SNS, Warszawa

63

63

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;
}
```

© UKSW, WMP, SNS, Warszawa

64

64

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 określonej przez parę iteratorów [*_First*, *_Last*) w innym porządku, tj. zamieniając lewą i prawą połowę elementów sekwencji miejscami, przy czym iterator *_Middle* wskazuje środek.

© UKSW, WMP, SNS, Warszawa

65

65

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;
}
```

© UKSW, WMP, SNS, Warszawa

66

66

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.

© UKSW, WMP, SNS, Warszawa

67

67

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;
}
```

© UKSW, WMP, SNS, Warszawa

68

68

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`.

© UKSW, WMP, SNS, Warszawa

69

69

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.

© UKSW, WMP, SNS, Warszawa

70

70

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;
}
```

© UKSW, WMP, SNS, Warszawa

71

71

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`.

© UKSW, WMP, SNS, Warszawa

72

72

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;
}
```

© UKSW, WMP, SNS, Warszawa

73

73

STL: algorytmy modyfikujące

Podsumowanie:

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

© UKSW, WMP, SNS, Warszawa

74

74

STL: algorytmy sortujące

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

© UKSW, WMP, SNS, Warszawa

75

75

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 predykatie, a następnie pozostałe, niespełniające warunku. Zwracany iterator wskazuje element graniczny, tj. pierwszy element niespełniający warunku z predykatu.
- Po zakończeniu działania uporządkowanie elementów względem siebie zarówno w pierwszej jak i w drugiej grupie nie musi być zachowane.

© UKSW, WMP, SNS, Warszawa

76

76

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); // 1 9 3 7 5 6 4 8 2
    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;
}
```

© UKSW, WMP, SNS, Warszawa

77

77

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 predykatie, 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?

© UKSW, WMP, SNS, Warszawa

78

78

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.

© UKSW, WMP, SNS, Warszawa

79

79

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 określonej 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.

© UKSW, WMP, SNS, Warszawa

80

80

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;
}
```

© UKSW, WMP, SNS, Warszawa

81

81

STL: algorytmy sortujące

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

- Z sekwencji elementów określonej przez parę iteratorów [*_First*, *_Last*) sortuje w porządku nierosnącym liczbę elementów, 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.

© UKSW, WMP, SNS, Warszawa

82

82

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

© UKSW, WMP, SNS, Warszawa

83

83

STL: algorytmy sortujące

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

- Z sekwencji elementów określonej przez parę iteratorów [*_First1*, *_Last1*) sortuje w porządku nierosnącym liczbę elementów, którą da się zmieścić w zakresie [*_First2*, *_Last2*). Posortowane elementy kopiuje 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.

© UKSW, WMP, SNS, Warszawa

84

84

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);
    // V: 1 2 3 4 5
    std::cout << "V zawiera:";
    for (std::vector<int>::iterator it=V.begin(); it!=V.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

85

85

STL: algorytmy sortujące

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

- *reorganizuje* sekwencję określaną przez parę iteratorów [*_First*, *_Last*) w porządku nierosnącym. To znaczy, że po posortowaniu:
 - element wskazywany przez *_Nth* zostaje zamieniony na element, który powinien znajdować się na tej pozycji, gdyby sekwencja [*_First*, *_Last*) była posortowana,
 - elementy mniejsze lub równe elementowi wskazywanemu przez *_Nth* umieszczane są w lewej części sekwencji, tj. [*_First*, *_Nth*),
 - elementy większe lub równe umieszczane są w prawej części, tj. [*_Nth*, *_Last*),
 - żadna z tych dwóch podsekwencji nie musi być posortowana.

- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

© UKSW, WMP, SNS, Warszawa

86

86

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

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

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

int main () {
    int myints[] = { 5, 10, 6, 4, 3, 2, 6, 7, 9, 3 };
    std::vector<int> v(myints, myints+10);
    std::nth_element(v.begin(), v.begin() + 5, v.end());
    // np. 3 2 3 4 5 6 10 7 9 6
    // albo 2 3 3 4 5 6 7 9 10
    // albo ..
    std::cout << "mediana: " << v[5] << std::endl;
    std::nth_element(v.begin(), v.begin() + 1, v.end(), porownaj);
    std::cout << "druga co do wielkości wartość: " << v[1] << std::endl;
    // np. 10 9 6 7 6 3 5 4 3 2
    // albo 10 9 7 6 6 5 4 3 3 2
    // albo ..
```

© UKSW, WMP, SNS, Warszawa

87

87

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.

© UKSW, WMP, SNS, Warszawa

88

88

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.

© UKSW, WMP, SNS, Warszawa

89

89

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 30 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;
}
```

© UKSW, WMP, SNS, Warszawa

90

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.

© UKSW, WMP, SNS, Warszawa

91

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()) << '\n';
    std::cout << " and " << (bounds.second - v.begin()) << '\n';
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

92

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.

© UKSW, WMP, SNS, Warszawa

93

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 identyczne 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 wejściowe
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

© UKSW, WMP, SNS, Warszawa

94

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;
}
```

© UKSW, WMP, SNS, Warszawa

95

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 $\text{distance}(\text{first1}, \text{last1}) + \text{distance}(\text{first2}, \text{last2}) - 1$
 - **inplace_merge**: dokładnie $N-1$ porównań, jeżeli dostępny jest wystarczający obszar pamięci, w przeciwnym razie: $N \cdot \log(N)$ gdzie $N = \text{distance}(\text{first}, \text{last})$.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

© UKSW, WMP, SNS, Warszawa

96

96

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.

© UKSW, WMP, SNS, Warszawa

97

97

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;
}
```

© UKSW, WMP, SNS, Warszawa

98

98

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

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

- sprawdza, czy pierwszy posortowany zakres `[_First1, _Last1)` zawiera wszystkie elementy z drugiego posortowanego zakresu `[_First2, _Last2)` (tj. czy drugi jest podzbiorem pierwszego – 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.

© UKSW, WMP, SNS, Warszawa

99

99

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;
}
```

© UKSW, WMP, SNS, Warszawa

100

100

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.

© UKSW, WMP, SNS, Warszawa

101

101

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;
}
```

© UKSW, WMP, SNS, Warszawa

102

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.

© UKSW, WMP, SNS, Warszawa

103

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;
}
```

© UKSW, WMP, SNS, Warszawa

104

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 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.

© UKSW, WMP, SNS, Warszawa

105

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;
}
```

© UKSW, WMP, SNS, Warszawa

106

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.

© UKSW, WMP, SNS, Warszawa

107

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;
}
```

© UKSW, WMP, SNS, Warszawa

108

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.

© UKSW, WMP, SNS, Warszawa

109

109

STL: wyszukiwanie w sekwencjach posortowanych i sortowanie

```
#include <iostream> // std::cout, std::boolalpha
#include <algorithm> // std::lexicographical_compare
#include <ctype> // 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 porównaj: ";
    std::cout << std::lexicographical_compare (V1,V1+5,V2,V2+9,porownaj);
    std::cout << '\n';
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

110

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 porządek na pierwszą permutację i zwraca `false`.
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

© UKSW, WMP, SNS, Warszawa

111

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;
}
```

© UKSW, WMP, SNS, Warszawa

112

112

STL: algorytmy sortujące

Podsumowanie

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

© UKSW, WMP, SNS, Warszawa

113

113