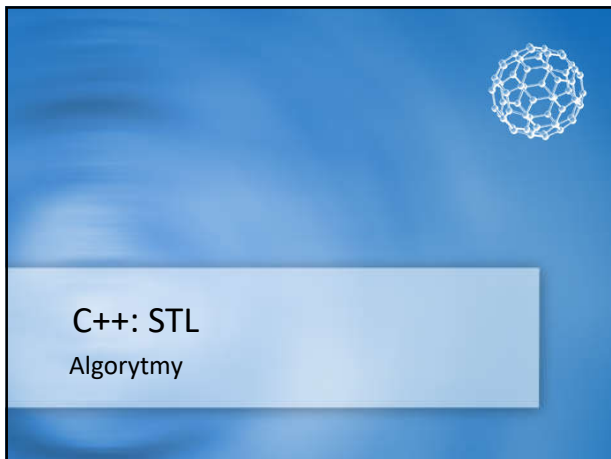




1

STL: algorytmy

Wprowadzenie

- W bibliotece STL znajduje się zbiór funkcji globalnych zwanych algorytmami, które ułatwiają działania na kontenerach
- `#include <algorithm>`
- Algorytmy dzielą się na cztery grupy:
 - niemodyfikujące (`for_each`, `find`, `count`, `equal`, `search`, etc.),
 - modyfikujące (`copy`, `swap`, `transform`, `replace`, `fill`, `generate`, `remove`, `unique`, `reverse`, `rotate`, `random_shuffle`),
 - sortujące (`partition`, `sort`, `merge`, operacje na zbiorach i na stercie, `min`, `max`),
 - inne, m.in. numeryczne.

© UKSW, WMP, SNS, Warszawa

2

STL: algorytmy niemodyfikujące

- `for_each`,
- `find`,
- `find_first_of`,
- `adjacent_find`,
- `count`,
- `count_if`,
- `mismatch`,
- `equal`,
- `search`,
- `find_end`,
- `search_n`

© UKSW, WMP, SNS, Warszawa

3

STL: algorytmy niemodyfikujące

```
template<typename InputIterator, typename Function>
Function for_each( InputIterator _First,
                  InputIterator _Last,
                  Function _Func );
```

- wywołuje podany obiekt funkcyjny na rzecz każdego z obiektów należących do sekwencji określonej przez parę iteratorów. Obiekty z sekwencji są przekazywane do obiektu funkcyjnego w argumentach wywołania. Iterator `_First` wskazuje na pierwszy element wskazanej sekwencji. Iterator `_Last` wskazuje na prawy graniczny element sekwencji – pierwszy, z listy zawartej w kontenerze, który do niej nie należy,
- zwraca kopię obiektu funkcyjnego po jego zastosowaniu.

© UKSW, WMP, SNS, Warszawa

4

STL: algorytmy niemodyfikujące

Obiekt funkcyjny:

```
template <class Type>
class GetValue
{
private:
    Type Factor;
    Type Res;
public:
    GetValue(const Type& _Val) :
        Factor(_Val), Res(0) {}
    void operator()(Type elem)
    {
        Res += Factor/elem;
    };
    Type _getRes() { return Res; }
};
```

```
int main()
{
    vector<int> v1;
    for (int i = 3; i < 7; i++)
        v1.push_back(i);

    GetValue<int> GV(1);
    GV = for_each(v1.begin(), v1.end(),
                 GV);

    cout << GV._getRes() << endl;
}
```

© UKSW, WMP, SNS, Warszawa

5

STL: algorytmy niemodyfikujące

Obiekt funkcyjny:

```
template <class Type>
class MultValue
{
private:
    Type Factor;
public:
    MultValue(const Type& _Val) :
        Factor(_Val) {}
    void operator()(Type& elem) const
    {
        elem *= Factor;
    };
};
```

```
int main()
{
    vector<int> v1;
    for (int i = -4; i <= 2; i++)
        v1.push_back(i);

    for_each(v1.begin(), v1.end(),
             MultValue<int>(-2));

    cout << "Mnożenie elementów z v1\n"
          << "przez -2 daje:\n"
          << endl;
    vector<int>::iterator Iter1;
    for (Iter1 = v1.begin();
         Iter1 != v1.end(); Iter1++)
        cout << *Iter1 << " ";
    cout << endl;
}
```

Znak &' konieczny, aby
- edytować elementy kontenera,
- uniknąć tworzenia kopii lokalnej
elementu przy wywołaniu

© UKSW, WMP, SNS, Warszawa

6

STL: algorytmy niemodyfikujące

```
template<typename InputIterator, typename Type>
InputIterator find( InputIterator _First,
                   InputIterator _Last,
                   const Type& _Val );
```

- w sekwencji `[_First, _Last)` znajduje położenie pierwszego elementu przechowującego określoną wartość `_Val`,
- zwraca iterator wskazujący na znaleziony element, a jeżeli takiego elementu nie znaleziono – iterator wskazujący na pierwszy element znajdujący się za ostatnim elementem z sekwencji, tj. iterator `_Last`.

Przykład:

Należy znaleźć w kontenerze element o wskazanej wartości.

© UKSW, WMP, SNS, Warszawa

7

7

STL: algorytmy niemodyfikujące

```
int main()
{
    list<int> L;
    list<int>::iterator Iter;
    list<int>::iterator result;

    L.push_back( 40 );
    L.push_back( 20 );
    L.push_back( 10 );
    L.push_back( 40 );
    L.push_back( 10 );
    result = find( L.begin(), L.end(), 10 );
    if ( result == L.end() )
        cout << "Nie ma 10 w liście L." << endl;
    else
        result++;
    cout << "Jest 10 w liście L. Po 10 występuje " << *(result) << "." << endl;
}
```

© UKSW, WMP, SNS, Warszawa

8

8

STL: algorytmy niemodyfikujące

```
template<typename ForwardIterator1,
        typename ForwardIterator2>
ForwardIterator1 find_first_of(
    ForwardIterator1 _First1, ForwardIterator1 _Last1,
    ForwardIterator2 _First2, ForwardIterator2 _Last2 );
```

- w sekwencji określonej przez parę iteratorów `[_First1, _Last1)` poszukuje pierwszego wystąpienia dowolnej z wartości ze zbioru wartości określanych przez parę iteratorów `[_First2, _Last2)`
- zwraca iterator wskazujący na znaleziony element, a jeżeli takiego elementu nie znaleziono – iterator wskazujący na pierwszy element znajdujący się za ostatnim elementem z sekwencji, tj. iterator `_Last1`,
- występuje w dwóch wersjach – druga wersja zawiera trzeci parametr szablonu reprezentujący obiekt funkcyjny służący do porównań i jest on podawany jako piąty argument.

© UKSW, WMP, SNS, Warszawa

9

9

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::find_first_of
#include <vector>    // std::vector
#include <ctype>     // std::tolower

int main() {
    int mychars[] = {'a','b','c','A','B','C'};
    std::vector<char> haystack (mychars, mychars+6);
    std::vector<char>::iterator it;
    int needle[] = {'A','B','C'};
    it = find_first_of (haystack.begin(), haystack.end(), needle, needle+3);

    if (it!=haystack.end())
        std::cout << "Pierwszy, który pasuje: " << *it << '\n';

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

10

10

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::find_first_of
#include <vector>    // std::vector
#include <ctype>     // std::tolower
bool comp_case_insensitive (char c1, char c2) {
    return (std::tolower(c1)==std::tolower(c2));
};

int main() {
    int mychars[] = {'a','b','c','A','B','C'};
    std::vector<char> haystack (mychars, mychars+6);
    std::vector<char>::iterator it;
    int needle[] = {'A','B','C'};
    it = find_first_of (haystack.begin(), haystack.end(),
                        needle, needle+3, comp_case_insensitive);
    if (it!=haystack.end())
        std::cout << "Pierwszy, który pasuje: " << *it << '\n';
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

11

11

STL: algorytmy niemodyfikujące

```
template<typename ForwardIterator>
ForwardIterator adjacent_find( ForwardIterator _First,
                              ForwardIterator _Last );
```

- w sekwencji określonej przez parę iteratorów `[_First, _Last)` poszukuje dwóch sąsiadujących ze sobą elementów, które są sobie równe,
- zwraca iterator wskazujący na pierwszy ze znalezionych par elementów, a jeżeli takiej pary nie znaleziono – iterator `_Last`,
- występuje w dwóch wersjach – druga wersja zawiera dodatkowy parametr szablonu reprezentujący obiekt funkcyjny służący do porównań.

© UKSW, WMP, SNS, Warszawa

12

12

STL: algorytmy niemodyfikujące

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

int main () {
    int myT[] = {5,20,5,30,30,20,10,10,20};
    std::vector<int> V (myT,myT+8);
    std::vector<int>::iterator it;

    it = std::adjacent_find (V.begin(), V.end());

    if (it!=V.end())
        std::cout << "Pierwszy z duplikatów: " << *it << '\n';

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

13

13

STL: algorytmy niemodyfikujące

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

bool myfunction (int i, int j) {
    return (i-j>10); // tu miejsce na nasz własny sposób porównywania
};

int main () {
    int myT[] = {5,20,5,30,30,20,10,10,20};
    std::vector<int> V (myT,myT+8);
    std::vector<int>::iterator it;

    it = std::adjacent_find (V.begin(), V.end(), myfunction);

    if (it!=V.end())
        std::cout << "Pierwszy z pary o dużej różnicy wartości: " << *it << '\n';

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

14

14

STL: algorytmy niemodyfikujące

```
template<typename InputIterator, typename Type>
typename iterator_traits<InputIterator>::difference_type
count( InputIterator _First,
       InputIterator _Last,
       const Type& _Val );
```

- w sekwencji określonej przez parę iteratorów `[_First, _Last]` zlicza elementy których wartości są równe wartości podanej w argumencie `_Val`,
- zwraca liczbę znalezionych elementów posługując się typem danych zdefiniowanym w cechach (traits) `difference_type`.

Typ `difference_type` służy do wyrażania wartości „odległość między iteratorami”. Powinien bezbłędnie konwertować się do `int`.

© UKSW, WMP, SNS, Warszawa

15

15

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::count
#include <vector> // std::vector
using namespace std;

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator iter;
    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 10 );
    v1.push_back( 40 );
    v1.push_back( 10 );

    int result;
    result = count( v1.begin(), v1.end(), 10 );
    cout << "Liczba wystąpień wartości 10 wynosi: " << result << ", " << endl;
}
```

© UKSW, WMP, SNS, Warszawa

16

16

STL: algorytmy niemodyfikujące

```
template<typename InputIterator, typename Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if( InputIterator _First,
          InputIterator _Last,
          Predicate _Pred );
```

- w sekwencji określonej przez parę iteratorów zlicza elementy, które spełniają warunek określony w predykanie (obiekt funkcyjny, funkcja),
- zwraca liczbę znalezionych elementów.

© UKSW, WMP, SNS, Warszawa

17

17

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::count_if
#include <vector> // std::vector
using namespace std;

bool greater10 ( int value ) {
    return value > 10; // tu miejsce na nasz własny sposób porównywania
};

int main() {
    vector<int> v1;
    vector<int>::iterator iter;
    v1.push_back( 10 );
    v1.push_back( 20 );
    v1.push_back( 10 );
    v1.push_back( 40 );
    v1.push_back( 10 );

    int result1 = count_if( v1.begin(), v1.end(), greater10 );
    cout << "Liczba wystąpień wartości większych od 10 w v1 wynosi: " <<
    result1 << ", " << endl;
}
```

© UKSW, WMP, SNS, Warszawa

18

18

STL: algorytmy niemodyfikujące

```
template<typename InputIterator1, typename InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(
    InputIterator1 _First1, InputIterator1 _Last1,
    InputIterator2 _First2 );
```

- porównuje dwie sekwencje elementów wskazywane przez parę iteratorów [_First1, _Last1) – pierwsza sekwencja, oraz przez iterator _First2 – druga sekwencja, odpowiednio element po elemencie poszukując pierwszej takiej pary elementów z obydwu sekwencji, dla których operator porównania zwróci wartość false (obiekty przechowywane w sekwencjach muszą mieć zaimplementowany przeciążony operator porównania),
- Zwraca parę iteratorów wskazującą na pierwszą znaną parę różniących się elementów z dwóch sekwencji, lub na parę pierwszych elementów tuż za sekwencjami, jeżeli sekwencje są identyczne,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny służący do porównań.

© UKSW, WMP, SNS, Warszawa

19

19

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::mismatch
#include <vector> // std::vector
#include <utility> // std::pair

int main () {
    std::vector<int> V;
    for (int i=1; i<6; i++)
        V.push_back (i*10); // V: 10 20 30 40 50
    int myT[] = {10,20,80,320,1024}; // myT: 10 20 80 320 1024
    std::pair<std::vector<int>::iterator,int*> mypair;

    mypair = std::mismatch (V.begin(), V.end(), myT);

    std::cout << " Pierwsza para różnych elementów: " << *mypair.first;
    std::cout << " oraz " << *mypair.second << '\n';
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

20

20

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::mismatch
#include <vector> // std::vector
#include <utility> // std::pair
bool mypredicate(double i, double j) {
    return (fabs(i-j) < 0.001); // tu miejsce na nasz własny sposób porównywania
};

int main() {
    std::vector<double> V;
    for (int i = 1; i < 6; i++)
        V.push_back(i * 10); // V: 10 20 30 40 50
    double myT[] = { 10,20,80,320,1024 }; // myT: 10 20 80 320 1024
    std::pair<std::vector<double>::iterator, double*> mypair;
    mypair = std::mismatch(V.begin(), V.end(), myT, mypredicate);
    std::cout << " Pierwsza para różnych elementów: " << *mypair.first;
    std::cout << " oraz " << *mypair.second << '\n';
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

21

21

STL: algorytmy niemodyfikujące

```
template<typename InputIterator1, typename InputIterator2>
bool equal( InputIterator1 _First1, InputIterator1 _Last1,
            InputIterator2 _First2 );
```

- porównuje dwie sekwencje elementów wskazywane przez parę iteratorów [_First1, _Last1) – pierwsza sekwencja, oraz iterator _First2 – druga sekwencja, odpowiednio element po elemencie
- zwraca wartość true, jeżeli sekwencje są identyczne, lub false – wpp,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyjny/funkcję służący do porównań.

© UKSW, WMP, SNS, Warszawa

22

22

STL: algorytmy niemodyfikujące

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

int main () {
    int myT[] = {20,40,60,80,100}; // myT: 20 40 60 80 100
    std::vector<int>V (myT,myT+5); // V: 20 40 60 80 100

    if ( std::equal (V.begin(), V.end(), myT) )
        std::cout << "Sekwencje identyczne.\n";
    else
        std::cout << "Sekwencje różne.\n";
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

23

23

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::equal
#include <vector> // std::vector
bool mypredicate (int i, int j) {
    return (i==j); // tu miejsce na nasz własny sposób porównywania
};

int main () {
    int myT[] = {20,40,60,80,100}; // myT: 20 40 60 80 100
    std::vector<int>V (myT,myT+5); // V: 20 40 60 80 100

    if ( std::equal (V.begin(), V.end(), myT, mypredicate) )
        std::cout << "Sekwencje identyczne.\n";
    else
        std::cout << "Sekwencje różne.\n";

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

24

24

STL: algorytmy niemodyfikujące

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
    ForwardIterator1 _First1, ForwardIterator1 _Last1,
    ForwardIterator2 _First2, ForwardIterator2 _Last2 );
```

- w sekwencji określonej przez parę iteratorów `[_First1, _Last1)` poszukuje pierwszej pod-sekwencji o wartościach identycznych jak sekwencja określana przez parę iteratorów `[_First2, _Last2)`,
- zwraca iterator na element w sekwencji określonej przez parę iteratorów `[_First1, _Last1)`, który rozpoczyna znaną pod-sekwencję, a jeżeli takiej pod-sekwencji nie znaleziono – iterator wskazujący na pierwszy element znajdujący się za ostatnim elementem z sekwencji `[_First1, _Last1)`, tj. iterator `_Last1`,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyny/funkcję służący do porównań.

© UKSW, WMP, SNS, Warszawa

25

25

STL: algorytmy niemodyfikujące

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
    ForwardIterator1 _First1, ForwardIterator1 _Last1,
    ForwardIterator2 _First2, ForwardIterator2 _Last2 );
```

- w sekwencji określonej przez parę iteratorów `[_First1, _Last1)` poszukuje ostatniej pod-sekwencji o wartościach identycznych jak sekwencja określana przez parę iteratorów `[_First2, _Last2)`,
- zwraca iterator na element w sekwencji określonej przez parę iteratorów `[_First1, _Last1)`, który rozpoczyna znaną pod-sekwencję, a jeżeli takiej pod-sekwencji nie znaleziono – iterator wskazujący na pierwszy element znajdujący się za ostatnim elementem z sekwencji `[_First1, _Last1)`, tj. iterator `_Last1`,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyny/funkcję służący do porównań.

© UKSW, WMP, SNS, Warszawa

26

26

STL: algorytmy niemodyfikujące

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

int main () {
    std::vector<int> haystack;
    // set some values:
    for (int i=1; i<10; i++) haystack.push_back(i*10);

    int needle1[] = {40,50,60,70};
    std::vector<int>::iterator it;
    it = std::search(haystack.begin(), haystack.end(), needle1, needle1+4);
    if (it!=haystack.end())
        std::cout << "needle1 found at position " <<(it-haystack.begin())<< '\n';
    else
        std::cout << "needle1 not found\n";

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

27

27

STL: algorytmy niemodyfikujące

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

bool mypredicate (int i, int j) {
    return (i==j); // tu miejsce na nasz własny sposób porównywania
}

int main () {
    std::vector<int> haystack;
    // set some values:
    for (int i=1; i<10; i++) haystack.push_back(i*10);
    int needle2[] = {20,30,50};
    it = std::search(haystack.begin(), haystack.end(), needle2, needle2+3,
        mypredicate);
    if (it!=haystack.end())
        std::cout << "needle2 found at position " <<(it-haystack.begin())<< '\n';
    else
        std::cout << "needle2 not found\n";

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

28

28

STL: algorytmy niemodyfikujące

```
template<class ForwardIterator1, class Diff2, class Type>
ForwardIterator1 search_n( ForwardIterator1 _First1,
    ForwardIterator1 _Last1,
    Size2 _Count, const Type& _Val );
```

- w sekwencji określonej przez parę iteratorów `[_First1, _Last1)` szuka pierwszej pod-sekwencji o długości `_Count`, w której wszystkie elementy mają wartość `_Val`
- zwraca iterator na element w sekwencji określonej przez parę iteratorów `[_First1, _Last1)`, który rozpoczyna znaną pod-sekwencję, a jeżeli takiej pod-sekwencji nie znaleziono – iterator wskazujący na pierwszy element znajdujący się za ostatnim elementem z sekwencji `[_First1, _Last1)`, tj. iteratorem `_Last1`,
- występuje w dwóch wersjach – druga z dodatkowym, ostatnim parametrem szablonu reprezentującym obiekt funkcyny/funkcję służący do porównań.

© UKSW, WMP, SNS, Warszawa

29

29

STL: algorytmy niemodyfikujące

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

int main () {
    int myT[]={10,20,30,30,20,10,10,20};
    std::vector<int> V (myT,myT+8);
    std::vector<int>::iterator it;
    it = std::search_n (V.begin(), V.end(), 2, 30);
    if (it!=V.end())
        std::cout << "two 30s found at position " <<(it-V.begin())<< '\n';
    else
        std::cout << "match not found\n";

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

30

30

STL: algorytmy niemodyfikujące

```
#include <iostream> // std::cout
#include <algorithm> // std::search_n
#include <vector> // std::vector
bool mypredicate (int i, int j) {
    return (i==j); // tu miejsce na nasz własny sposób porównywania
};
int main () {
    int myT[]={10,20,30,30,20,10,10,20};
    std::vector<int> V (myT,myT+8);
    std::vector<int>::iterator it;
    it = std::search_n (V.begin(), V.end(), 2, 10, mypredicate);
    if (it!=V.end())
        std::cout<<"two 10s found at position "<<int(it-V.begin())<< '\n';
    else
        std::cout << "match not found\n";
    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

31

31

STL: algorytmy niemodyfikujące

Podsumowanie:

1. `for_each`,
2. `find`,
3. `binary_search`,
4. `find_first_of`,
5. `adjacent_find`,
6. `count`,
7. `count_if`,
8. `mismatch`,
9. `equal`,
10. `search`,
11. `find_end`,
12. `search_n`.

© UKSW, WMP, SNS, Warszawa

32

32

STL: algorytmy modyfikujące

- | | |
|------------------------------------|-----------------------------------|
| 1. <code>fill</code> , | 14. <code>reverse</code> , |
| 2. <code>fill_n</code> , | 15. <code>reverse_copy</code> , |
| 3. <code>generate</code> , | 16. <code>rotate</code> , |
| 4. <code>generate_n</code> , | 17. <code>rotate_copy</code> , |
| 5. <code>copy</code> , | 18. <code>random_shuffle</code> , |
| 6. <code>copy_backward</code> , | 19. <code>transform</code> |
| 7. <code>swap</code> , | |
| 8. <code>swap_ranges</code> , | |
| 9. <code>replace</code> , | |
| 10. <code>replace_if</code> , | |
| 11. <code>replace_copy</code> , | |
| 12. <code>replace_copy_if</code> , | |
| 13. <code>unique_copy</code> , | |

© UKSW, WMP, SNS, Warszawa

33

33

STL: algorytmy modyfikujące

```
template<typename ForwardIterator, typename Type>
void fill( ForwardIterator _First, ForwardIterator _Last,
          const Type& _Val );
```

- w sekwencji określonej przez parę iteratorów [`_First`, `_Last`) zmienia wartości przechowywane w elementach tej sekwencji na wartość podaną w argumencie `_Val`.

© UKSW, WMP, SNS, Warszawa

34

34

STL: algorytmy modyfikujące

```
template<typename OutputIterator, typename Size,
        typename Type>
void fill_n( OutputIterator _First, Size _Count,
            const Type& _Val );
```

- w pierwszych `_Count` elementach sekwencji, której pierwszy element jest wskazywany przez iterator `_First`, zmienia wartości przechowywane w tych elementach na wartość podaną w argumencie `_Val`.

© UKSW, WMP, SNS, Warszawa

35

35

STL: algorytmy modyfikujące

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

int main () {
    std::vector<int> V (8); // V: 0 0 0 0 0 0 0 0

    std::fill (V.begin(), V.begin()+4, 5); // V: 5 5 5 5 0 0 0 0
    std::fill (V.begin()+3, 3, 8); // V: 5 5 8 8 8 8 0 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

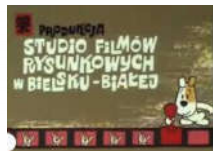
36

36

STL: algorytmy modyfikujące

```
template<typename ForwardIterator, typename Generator>
void generate( ForwardIterator _First, ForwardIterator _Last,
              Generator _Gen );
```

- w sekwencji określonej przez parę iteratorów `[_First, _Last)` wartości przechowywane w elementach tej sekwencji zastępuje wartościami wygenerowanymi przez obiekt funkcyjny `_Gen` (stosowany jest operator przypisania). Wywołanie `_Gen` jest bezargumentowe, tj. wartość elementu z sekwencji nie jest uwzględniana przez `_Gen`.



© UKSW, WMP, SNS, Warszawa

37

37

STL: algorytmy modyfikujące

```
int Fibonacci(void)          int main()
{
    static int r;            const int VECTOR_SIZE = 8 ;
    static int f1 = 0;        typedef vector<int> IntVector ;
    static int f2 = 1;        typedef IntVector::iterator IntVectorIt ;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
    return f1 ;
};

// zmienne statyczne
// deklarowane są tylko raz

IntVector Numbers(VECTOR_SIZE) ;
IntVectorIt start, end, it ;
start = Numbers.begin() ;
end = Numbers.end() ;
generate(start, end, Fibonacci) ;

cout << "Liczby { " ;
for(it = start; it != end; it++)
    cout << *it << " " ;
cout << " }\n" << endl ;
return 0 ;
};

Wyjście: Liczby { 1 1 2 3 5 8 13 21 }
```

© UKSW, WMP, SNS, Warszawa

38

38

STL: algorytmy modyfikujące

```
template<class OutputIterator, class Size, class Generator>
void generate_n( OutputIterator _First, Size _Count,
                Generator _Gen );
```

- w pierwszych `_Count` elementach sekwencji, której pierwszy element jest wskazywany przez iterator `_First`, zmienia wartości przechowywane w elementach tej sekwencji na wartości wygenerowane przez obiekt funkcyjny `_Gen`.

© UKSW, WMP, SNS, Warszawa

39

39

STL: algorytmy modyfikujące

Porównanie działania `generate` i `for_each`:

generate:

```
{ // replace [_First, _Last) with _Func()
  for (; _First != _Last; ++_First)
    *_First = _Func();
}
```

for_each:

```
{ // perform function for each element
  for (; _First != _Last; ++_First)
    _Func(*_First);
}
```

© UKSW, WMP, SNS, Warszawa

40

40

STL: algorytmy modyfikujące

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy(
    InputIterator _First, InputIterator _Last,
    OutputIterator _DestBeg );
```

- Przepisuje wartości elementów z sekwencji źródłowej określonej przez parę iteratorów `[_First, _Last)` do sekwencji docelowej, której pierwszy element wskazywany jest przez iterator `_DestBeg`.

Uwaga: przedziały muszą być poprawnie zdefiniowane, w przedziale docelowym musi być dostatecznie dużo elementów; ponieważ algorytm kopiuje sekwencyjnie zaczynając od pierwszego elementu, przedziały mogą się nakładać – ważne, aby element wskazywany przez „_First” nie należał do przedziału docelowego.

© UKSW, WMP, SNS, Warszawa

41

41

STL: algorytmy modyfikujące

```
template<
    typename BidirectionalIterator1,
    typename BidirectionalIterator2>
```

```
BidirectionalIterator2 copy_backward(
    BidirectionalIterator1 _First,
    BidirectionalIterator1 _Last,
    BidirectionalIterator2 _DestEnd );
```

- przepisuje wartości elementów z sekwencji źródłowej określonej przez parę iteratorów `[_First, _Last)` do sekwencji docelowej, dla której pierwszy element znajdujący się za ostatnim z sekwencji docelowej wskazywany jest przez iterator `_DestEnd`.

Uwaga: wszystkie iteratory muszą być dwukierunkowe; przedziały muszą być poprawnie zdefiniowane, w przedziale docelowym musi być dostatecznie dużo elementów; ponieważ algorytm kopiuje sekwencyjnie zaczynając od pierwszego elementu, przedziały mogą się nakładać – ważne, aby element wskazywany przez „_First” nie należał do przedziału docelowego.

© UKSW, WMP, SNS, Warszawa

42

42

STL: algorytmy modyfikujące

Aby przesunąć sekwencję elementów np. o jeden w lewo należy użyć `copy`, natomiast o jeden w prawo – `copy_backward`, np.:

```
string s("abcdefghijklmnopqrstuvwxyz");
vector<char> vector1(s.begin(), s.end());

copy(vector1.begin() + 1, vector1.end(), vector1.begin());

copy_backward(vector1.begin(), vector1.end() - 1, vector1.end());
```

© UKSW, WMP, SNS, Warszawa

43

43

STL: algorytmy modyfikujące

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

int main () {
    std::vector<int> V;

    for (int i=1; i<=5; i++)
        V.push_back(i*10); // V: 10 20 30 40 50
    V.resize(V.size()+3); // allocate space for 3 more elements

    std::copy_backward ( V.begin(), V.begin()+5, V.end() );

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

W oknie konsoli:
V zawiera: 10 20 30 10 20 30 10 20 30 40 50

© UKSW, WMP, SNS, Warszawa

44

44