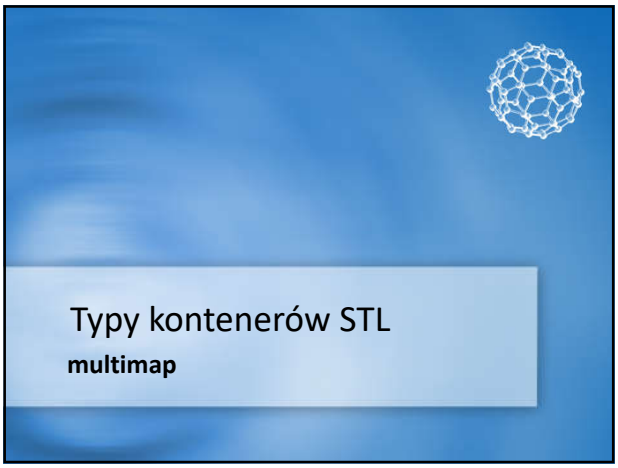


97

STL: kontenery

Kontenery asocjacyjne – **`multimap`**

realizuje funkcjonalność kontenera typu **`map`**, ale pozwala na przechowywanie wielu elementów o tej samej wartości klucza.

To może być przydatne, kiedy realizujemy funkcjonalność w rodzaju np. książki telefonicznej, gdzie jedna osoba może mieć przypisanych kilka numerów telefonu.

Przykład:

Dana jest baza osób i ich numerów telefonów (bywa, że osoba ma kilka telefonów). Należy wypisać na ekranie wszystkie numery telefonów żądanej osoby.

98

STL: kontenery

```
multimap<string, string> directory;
directory.insert(pair<string, string>("T","555-4533"));
directory.insert(pair<string, string>("T","555-9999"));
directory.insert(pair<string, string>("C","555-9678"));
string str;
cout << "Podaj imię: ";
cin >> str;
multimap<string, string>::iterator p = directory.find(str);
if(p != directory.end()) {
  do {
     cout << "Numer telefonu: " <<  p->second << endl;
     p++;
  } while(p != directory.upper_bound(str));  // pierwszy element,
                                             // którego klucz jest większy
}
else
    cout << "Nie ma takiego użytkownika telefonu.\n";
```


99

STL: kontenery

```
multimap<string, string> directory;
directory.insert(pair<string, string>("T","555-4533"));
directory.insert(pair<string, string>("T","555-9999"));
directory.insert(pair<string, string>("C","555-9678"));
string str;
cout << "Podaj imię: ";
cin >> str;
pair<multimap<string, string>::iterator, multimap<string, string>::iterator> ii;
ii = directory.equal_range(str);
for(multimap<string, string>::iterator it = ii.first; it != ii.second; ++it)
{
    cout<<"Klucz = "<<it->first<<"    Wart = "<<it->second<<endl;
}
```

`ii.second` *wskazuje na pierwszy element po znalezionej sekwencji elementów* **`str`**.

Stosowanie **`equal_range`** *to pewniejsze rozwiązanie, ponieważ specyfikacja STL nie mówi, że* **`find`** *powinien zawsze dać pierwszy element w serii, ale ten najwcześniej dodany.*

100

STL: kontenery

Przykład – usuwanie zapisów dla wskazanego użytkownika:

```
multimap<string, string> directory;
directory.insert(pair<string, string>("T","555-4533"));
directory.insert(pair<string, string>("T","555-9999"));
directory.insert(pair<string, string>("C","555-9678"));
string str;
cout << "Podaj imię: ";
cin >> str;
pair<multimap<string, string>::iterator, multimap<string, string>::iterator> ii;
ii = directory.equal_range(str);
directory.erase(ii.first, ii.second);
```

Ponieważ może być wiele zapisów dla tego samego klucza, w **`multimap`** nie ma przeciążonego operatora [], bo nie zawsze mógłby zwracać jeden element – czasem wynikiem jest sekwencja elementów. Stosowanie [] byłoby nieintuicyjne.

W **`multimap`** funkcjonalnym odpowiednikiem [] jest **`equal_range`**.

101

STL: kontenery

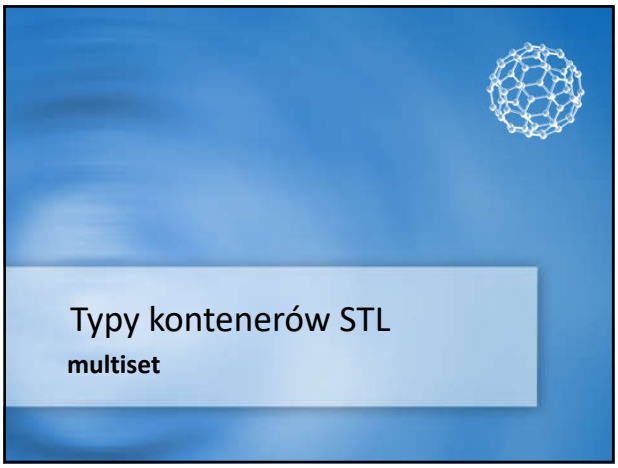
Kontenery asocjacyjne – **`multimap`**

Kontener **`multimap`** daje funkcjonalność, którą można uzyskać również w inny sposób, np. implementując mapę wektorów bądź mapę list.

Argumenty za **`multimap`**:

1. Wektory od razu rezerwują większy blok pamięci, dla zapewnienia jej ciągłości (szybkość dostępu).
2. Czas dostępu do elementów w strukturze kontenera kontenerów jest większy, co ma znaczenie np. przy wielokrotnie powtarzanych odczytach danych z multimapy, chociaż..
   przy efektywnej optymalizacji jest możliwość doprowadzenia do stanu, gdzie odczyt mapy kontenerów deque jest szybszy (ale zapis nie).

102



Typy kontenerów STL

multiset

103

STL: kontenery

Kontener **`multiset`**

- umożliwia przechowywanie obiektów o powtarzających się wartościach,
- wszystkie duplikaty są przechowywane w bezpośrednim sąsiedztwie.

W przykładzie na następnym slajdzie kontener będzie służył do zliczania wystąpień słów w pliku

104

STL: kontenery

```
int main(int argc, char* argv[]) {
   const char* fname = "main29.cpp";
   if(argc > 1) fname = argv[1];
   ifstream in(fname);
   multiset<string> wordmset;
   string word;
   while(in >> word)
        wordmset.insert(word);
   typedef multiset<string>::iterator MSit;
   MSit it = wordmset.begin();
   while(it != wordmset.end()) {
        pair<MSit, MSit> p = wordmset.equal_range(*it);
        int count = distance(p.first, p.second);    // funkcja globalna
                                                    // z biblioteki <iterator>
        cout << *it << ": " << count << endl;
        it = p.second; // przesuwamy iterator do następnego słowa
   }
```


105

STL: kontenery

Kontenery sekwencyjne vs asocjacyjne

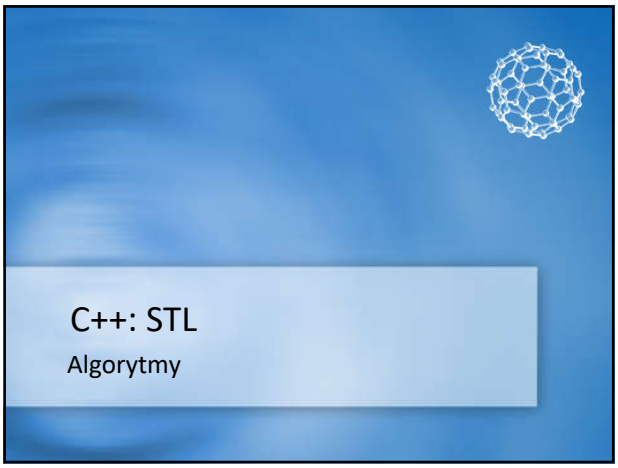
Sekwencyjne – implementowane jako tablice lub listy:

- Dodawanie elementu – stały czas
- Wstawianie elementu w środek – powolne

Asocjacyjne – implementowane jako drzewa binarne:

- Wstawianie elementu – O(log n)
- Usuwanie elementu – O(log n)
- Szukanie elementu – O(log n)
- Inkrementacja/dekrementacja iteratora – 1
- Wstawianie elementu w środek – szybkie

106



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:
  1. **niemodyfikujące** (`for_each, find, count, equal, search,` etc.),
  2. **modyfikujące** (`copy, swap, transform, replace, fill, generate, remove, unique, reverse, rotate, random_shuffle`),
  3. **sortujące** (`partition, sort, merge, operacje na zbiorach i na stercie, min, max`),
  4. **inne, m.in. numeryczne**.

2

STL: algorytmy niemodyfikujące

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

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 określanej przez parę iteratorów. Obiekty z sekwencji są przekazywane do obiektu funkcyjnego w argumencie 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.

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;
```


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;
  };
};
```

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

```
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" ;
    vector <int>::iterator Iter1;
    for ( Iter1 = v1.begin( );
          Iter1 != v1.end( ); Iter1++ )
       cout << *Iter1 << " ";
    cout << endl;
```


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.

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;
```


STL: algorytmy niemodyfikujące

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

- w sekwencji określanej przez parę iteratorów **[***_First1*,*_Last1***)** poszukuje pierwszego wystąpienia dowolnej z wartości ze zbioru wartości określanego 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.

STL: algorytmy niemodyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find_first_of
#include <vector>       // std::vector
#include <cctype>       // 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;
}
```


STL: algorytmy niemodyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find_first_of
#include <vector>       // std::vector
#include <cctype>       // 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;
}
```


STL: algorytmy niemodyfikujące

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

- w sekwencji określanej przez parę iteratorów **[***_First*,*_Last***)** poszukuje dwóch sąsiadujących ze sobą elementów, które są sobie równe,
- zwraca iterator wskazujący na pierwszy ze znalezionej pary 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.

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;
}
```


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;
}
```


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 określanej 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`**.

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;
}
```


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

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;
}
```


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 znalezioną 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.

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;
}
```


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;
}
```


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.

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;
}
```


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;
}
```


24

STL: algorytmy niemodyfikujące

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

- w sekwencji określanej 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 określanej przez parę iteratorów [_First1,_Last1), który rozpoczyna znalezioną 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 funkcyjny/funkcję służący do porównań.

STL: algorytmy niemodyfikujące

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

- w sekwencji określanej 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 określanej przez parę iteratorów [_First1,_Last1), który rozpoczyna znalezioną 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 funkcyjny/funkcję służący do porównań.

STL: algorytmy niemodyfikujące

```
#include <iostream>     // std::cout
#include <algorithm>    // std::search
#include <vector>       // std::vector
int main () {
  std::vector<int> haystack;
  // set some values:         haystack: 10 20 30 40 50 60 70 80 90
  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;
}
```


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:         haystack: 10 20 30 40 50 60 70 80 90
  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;
}
```


STL: algorytmy niemodyfikujące

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

- w sekwencji określanej 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 określanej przez parę iteratorów [_First1,_Last1), który rozpoczyna znalezioną 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 funkcyjny/funkcję służący do porównań.

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;
}
```

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;
}
```


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.**

STL: algorytmy modyfikujące

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,**
15. **reverse_copy,**
16. **rotate,**
17. **rotate_copy,**
18. **random_shuffle,**
19. **transform**

STL: algorytmy modyfikujące

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

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

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*.

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 5 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;
}
```

STL: algorytmy modyfikujące

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

- w sekwencji określanej 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`.

STL: algorytmy modyfikujące

```
int Fibonacci(void)
{
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
    return f1 ;
};

// zmienne statyczne
// deklarowane są tylko raz
```

```
int main()
{
    const int VECTOR_SIZE = 8 ;
    typedef vector<int> IntVector ;
    typedef IntVector::iterator IntVectorIt ;

    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 }`

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`.

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);
}
```


STL: algorytmy modyfikujące

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

- Przepisuje wartości elementów z sekwencji źródłowej określanej 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.

STL: algorytmy modyfikujące

```
template<   typename BidirectionalIterator1,
            typename BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
            BidirectionalIterator1 _First,
            BidirectionalIterator1 _Last,
            BidirectionalIterator2 _DestEnd );
```

- przepisuje wartości elementów z sekwencji źródłowej określanej 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.

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("abcdefghihklmnopqrstuvwxyz");
vector<char> vector1(s.begin(), s.end());

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

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


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 40 50
```


STL: algorytmy modyfikujące

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

- zamienia miejscami dwie wartości wskazywane przez dwa iteratory.

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`.

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 ( ) );
```