# 9. Итератори

4.2 Итератори.Същност и класификация
4.2.1 Входни и изходни итератори
4.2.2 Нарастващи итератори
4.2.3 Двупосочни итератори
4.2.4 Итератори с произволен достъп
4.2.5 Адаптери на итератори
4.3. Списък (list)
4.4. Двойно свързан списък (deque)

# 4.2 Итератори.Същност и класификация

Дефиниция:

Итераторите са обобщение на указателите:

- Те са обекти, които сочат други обекти;
- Итераторите позволяват да се итерират (преминават) елементи;
- В определен момент итераторът сочи един обект от контейнера, в следващият момент следващият и т.н.
- Позволяват на програмиста да работи по еднакъв начин с различни обекти, съхранени в контейнери.

# 4.2 Итератори.Същност и класификация

Основни член функции – зависят от типа на итератора:

- Конструктори;
- operator* който връща стойността на данните value type , сочени от итератора. Служи за достъп (четене/запис) на обектите в контейнера;
- operator= за получаване на стойност на самият итератор;
- operator== за сравнение с итератор;
- operator++ operator-- за нарастване/намаляване на стойността на итератора;

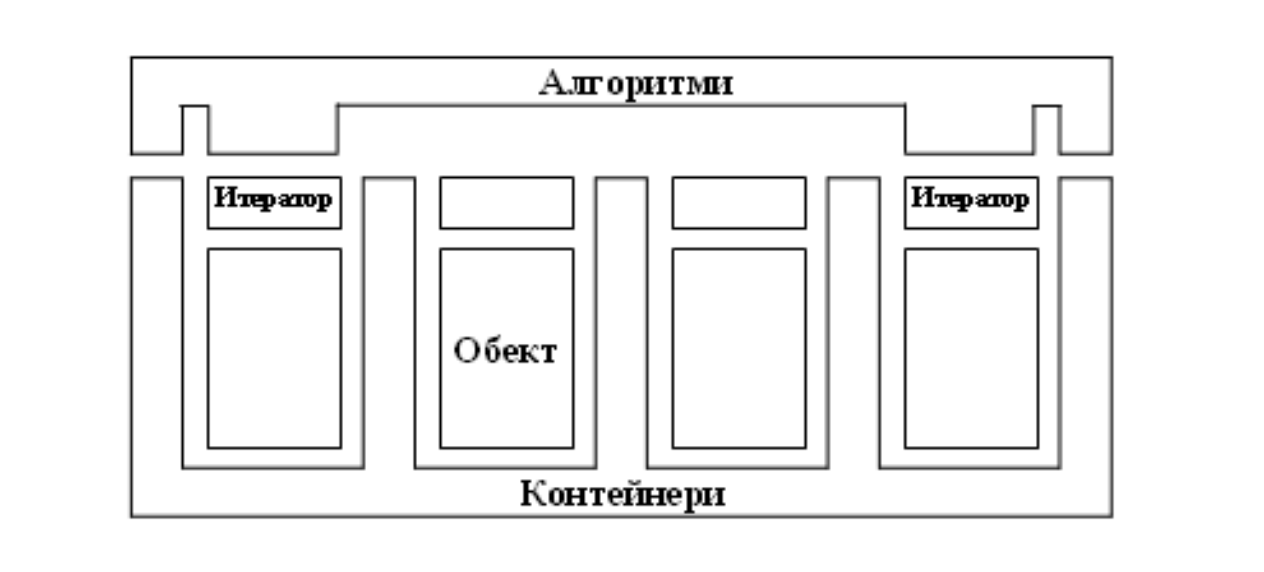
# 4.2 Итератори.Същност и класификация

- Предназначение:

  Итераторите са елемент на обобщеното програмиране и обобщените алгоритми-стил на програмиране
- Като аргументи на обобщените алгоритми;
- Итераторите определят начина, по който ще се осъществи достъпът до обектите в контейнера;
- Обобщение на указателите
  - всяка темплейтна функция, която получава итератори като аргументи работи и с нормалните указатели.

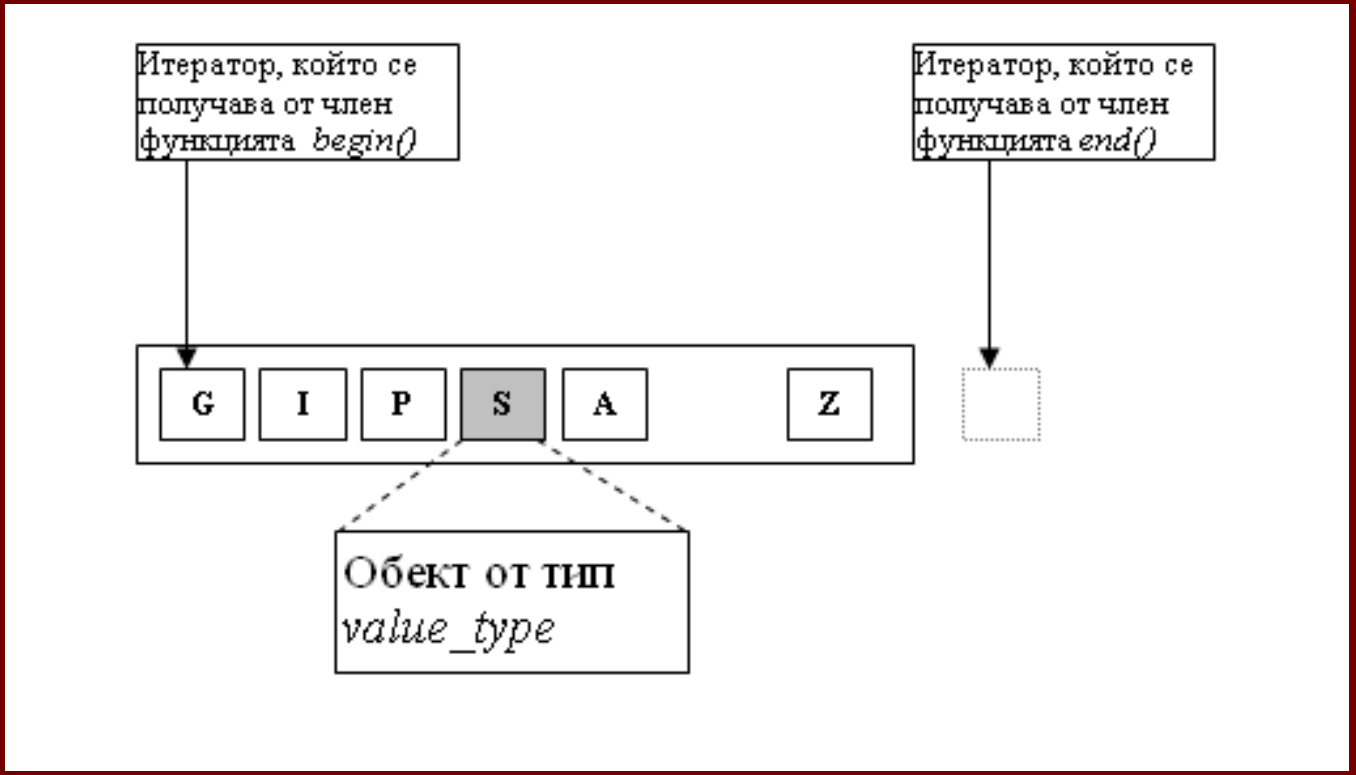
# 4.2 Итератори.Същност и класификация

Итераторите са сързващото звено между контейнерите и алгоритмите.
Схема на мястото в ортогоналната декомпозиция на компонентното пространство.



# 4.2 Итератори.Същност и класификация

Итераторите  към обектите на контейнера се получават от самия контейнер в резултат от изпълнението на член функции



# 4.2 Итератори.Същност и класификация

Пример за използване на конструктор, operator* и operator =:

```
vector<int> iv(3);
iv[0] = 3;
iv[1] = 2;
iv[2] = 6;
vector<int>::iterator first = iv.begin(); // конструктор и
vector<int>::iterator last  = iv.end(); // =
while (first != last) {
   cout << *first++ << " "; // използване на *
}
```

Изход: 3  2  6

# 4.2 Итератори.Същност и класификация

Пример (вариант C++11 с използване на auto за получаване на типа)

```
vector<int> iv(3);
iv[0] = 3;
iv[1] = 2;
iv[2] = 6;
auto first = iv.begin(); // конструктор и
auto last  = iv.end(); // =
while (first != last) {
   cout << *first++ << " "; // използване на *
}
```

Изход: 3  2  6

# 4.2 Итератори.Същност и класификация

Примерна програма за използването на итератори за обобщени алгоритми:

Алгоритъм за сортиране:

sort(<начален_итератор>,<итератор_след_края>)

Програмиране на приложение с използване на алгоритъма за сортиране:

```
#include <algorithm>
// създаване на вектора v
// запълване на вектора v
// прилагане на алгоритъм

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

# 4.2 Итератори.Същност и класификация

Пример за създаване на вектор чрез итератори:

```
vector<float> v (5, 3.25);
vector<float> v_new1 (v);  // конструира v_new1 чрез v
vector<float> v_new2 = v;  // присвоява v на vnew2
// конструира v_new3 чрез итерационен интервал от
// елементите на вектора v в частност целия
// контейнер:
vector<float> v_new3 (v.begin(), v.end() );
```

# 4.2 Итератори.Същност и класификация

**Примерна програма за проверка на еквивалентност между итератори (operator ==):**

```
auto vit1 = v.begin();
auto vit2 = v.begin()+2;
 (vit1 == vit2) ? cout << "equal" : cout << "different";
```

Изход: different

**Примерна програма за използване на член функцията insert(...) на вектор, която изисква итератор и елемент като аргументи :**

```
vector<int> iv(2,6);                    // вектор iv: 6 6
 iv.insert(iv.begin(), 2, 5);           // вектор iv: 5 5 6 6
 vector<int> ivect(1, 3);
 ivect.insert(ivect.end(), iv.begin(), iv.begin()+3);//вектор ivect: 3 5 5 6
```

# 4.2 Итератори.Същност и класификация

Пример за използване на итератори като параметри на член функциите на вектор:

Изтриването в края на вектор изисква константно време за изпълнение, а от средата(началото) - линейно време.

```
vector<float> iv (4, 8.0); // вектор iv:  8.0  8.0  8.0  8.0
iv.erase (iv.begin() );       // вектор iv:  8.0  8.0  8.0
iv.erase (iv.begin()+1 );         // вектор iv:  8.0  8.0
iv.erase (iv.begin(), iv.end() );// вектор iv е празен
```

# 4.2 Итератори.Същност и класификация

**Пример с начална инициализация ( C++ 11):**

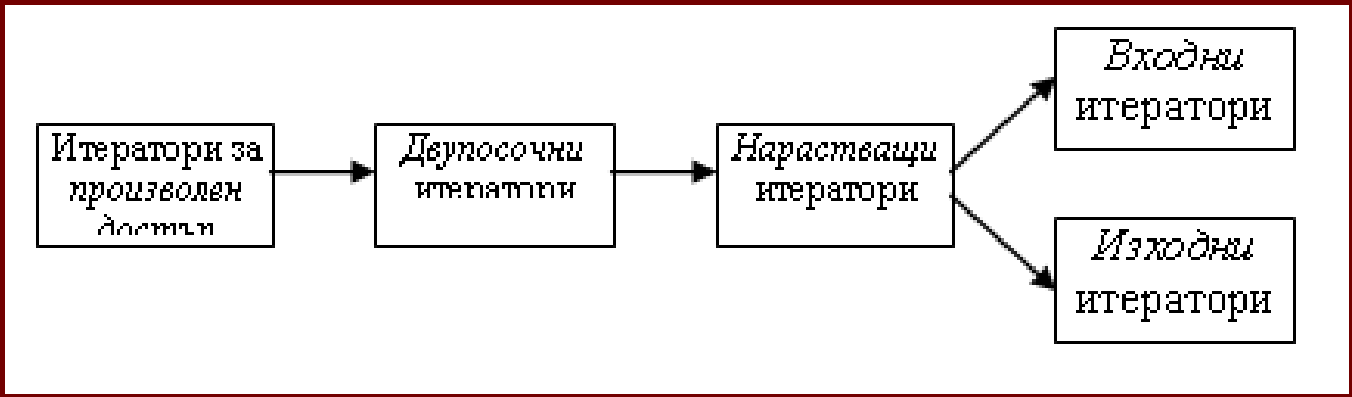
```
vector<float> iv{ 1., 2., 3., 4. };// вектор iv:  1.0  2.0  3.0  4.0
iv.erase(iv.begin());       // вектор iv:  2.0  3.0  4.0
iv.erase(iv.begin() + 1);       // вектор iv:  2.0  4.0
iv.erase(iv.begin(), iv.end());// вектор iv е празен
```

# 4.2 Итератори.Същност и класификация

| ***Контейнер*** | ***операция*** | ***Валидност на итератора*** |
|---|---|---|
| vector | добавяне | Необходима реалокация - всички итератори са невалидни |
| | | Няма реалокация - всички итератори преди позицията на добавянето остават валидни |
| | изтриване | всички итератори след позицията на изтриване стават невалидни |
| list | добавяне | всички итератори остават валидни |
| | изтриване | Само итератори към изтриваните елементи стават невалидни |
| deque | добавяне | всички итератори стават невалидни |
| | изтриване | всички итератори стават невалидни |

# 4.2 Итератори.Същност и класификация

Класификация: Пет типа итератори според операциите, дефинирани за тях:
Класова йерархия на итераторите:




Всеки итератор е проектиран да удовлетворява точно пределено множество от изисквания.

# 4.2 Итератори.Същност и класификация

<table>
<tr><th colspan="4">Категория</th><th>Свойства</th><th>Операции</th></tr>
<tr><td colspan="4">Всички</td><td>Копиращ конструктор, оператор присвояване<br>Унищожаване</td><td>X b(a);<br>b = a;</td></tr>
<tr><td colspan="4">Всички</td><td>Допуска увеличение</td><td>++a<br>a++</td></tr>
<tr><td rowspan="8">Random Access</td><td rowspan="5">Bidirectional</td><td rowspan="4">Forward</td><td>Input</td><td>Сравнение за еквивалентност/неекивалентност</td><td>a == b<br>a != b</td></tr>
<tr><td></td><td>Дереферииране в дясна страна rvalue</td><td>*a<br>a->m</td></tr>
<tr><td>Output</td><td>Дереферииране в лява страна lvalue<br>(само за типове mutable iterator)</td><td>*a = t<br>*a++ = t</td></tr>
<tr><td rowspan="2"></td><td>подразбиращ се конструктор</td><td>X a;<br>X()</td></tr>
<tr><td>Многопасов, т.е. дереферирането и инкрементирането не са свързани</td><td>{ b=a;<br>*a++; *b; }</td></tr>
<tr><td colspan="2"></td><td>Добавено декрементиране</td><td>--a<br>a--<br>*a--</td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Аритметика ( + и -) с указатели</td><td>a + n<br>n + a<br>a - n<br>a - b</td></tr>
<tr><td>Допуска допълнителни сравнители между итераторите (освен ==) т. е. (<, >, <= and >=)</td><td>a < b<br>a > b<br>a <= b<br>a >= b</td></tr>
<tr><td>Допуска увеличаване/намаляване с константа += и −=</td><td>a += n<br>a −= n</td></tr>
<tr><td>Допуска оператор за индексиране при дереферирането ([])</td><td>a[n]</td></tr>
</table>

# 4.2.1 Всички итератори

Всички итератори – член функции:

- Копиращ конструктор
- Оператор за увеличаване (оператор ++) в префиксен и постфиксен вариант:

  ++It

  It++

# 4.2.1 Входни и изходни итератори

Входен итератор – допълнителни член функции:

- Оператор за присвояване (=)
- Оператор за сравнение за еквивалентност/не еквивалентност (== и !=)
- Оператор за дереферииране (оператор *) Особеност:
  - Използва се само в дясната страна на операцията присвояване (***rvalue*** );

# 4.2.1 Изходни итератори

Изходен итератор – член функции:

- Оператор за присвояване (assignment operator = )
- Оператор за дереферииране (operator *)

  Особености:
  - Използва се само ***в лява страна lvalue (само за типове mutable iterator)***;
  - Подразбиращо се поведение “увеличение” и извежда следващия обект.

  Примери:

  std::ostream_iterator

  std::ostreambuf_iterator

  std::insert_iterator

  std::back_insert_iterator

  std::front_insert_iterator

# 4.2.1 Входни и изходни итератори

Основно предназначение – да се обработва информацията от вх/изх потоци:

- Вх./изх. се разглежда като контейнер

Входни и изходни итератори за обслужване на входно/изходните потоци - istream и ostream. Това са специалните итератори – istream_iterator и ostream_iterator.

# 4.2.1 Входни итератори

Основно предназначение на входния итератор:

Входни итератори за обслужване на входните потоци (istream):

**istream_iterator**

Подразбиращо се поведение “увеличение” и достъп до следващия входен обект.

- Четене;
- Увеличаване с 1 (++).

# 4.2.1 Входни и изходни итератори

Декларация на входен потоков итератор в STL:

```
template<class U, class Ch = char, class Tr = char_traits<Ch> >
    class istream_iterator
        : public iterator<input_iterator_tag, U, ptrdiff_t> {
public:
   typedef Ch char_type;
   typedef Tr traits_type;
   typedef basic_istream<Ch, Tr> istream_type;
   istream_iterator();
   istream_iterator(istream_type& is);
    const U& operator*() const;
    const U *operator->() const;
    istream_iterator<U, Ch, Tr>& operator++();
    istream_iterator<U, Ch, Tr> operator++(int);
    };
```

# 4.2.1 Входни и изходни итератори

Декларация в библиотеката на изходния потоков итератор

```
template<class U, class E=char, class Tr=char_traits<E> >
    class ostream_iterator
        : public iterator<output_iterator_tag, void, void> {
public:
    typedef U value_type;
    typedef E char_type;
    typedef T traits_type;
    typedef basic_ostream<E, T> ostream_type;
    ostream_iterator(ostream_type& os);
    ostream_iterator(ostream_type& os, const E *delim);
    ostream_iterator<U, E, T>& operator=(const U& val);
    ostream_iterator<U, E, T>& operator*();
    ostream_iterator<U, E, T>& operator++();
    ostream_iterator<U, E, T> operator++(int);
protected:
        const _E *_Delim;
    ostream_type *_Ostr;
};
```

# 4.2.1 Входни и изходни итератори

Примери:

Даден е файл със записани поредици от символи 0 и 1. Трябва да се прочетат от файла и да се изведат на конзолата (cout):

```
ifstream ifile ("example_file.txt");  // съдържа 110101110111011
char tmpChar;
while (ifile >> tmpChar) cout << tmpChar;
```

Изход (пример): 110101110111011

# 4.2.1 Входни и изходни итератори

Реализация чрез итератори и **алгоритъма copy**:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator  last, OutputIterator
    result);
#include <iostream> //  cout
#include <fstream>  //  ifstream
#include <iterator>  //  iterator's
#include <algorithm> //  copy
...
ifstream ifile("example_file.txt");
copy (istream_iterator<char> (ifile),    //конструиране на  нач. итератор
istream_iterator<char> (),                //конструиране  кр. итератор
ostream_iterator<char> (cout) );  //конструиране на  нач. итератор
```

# 4.2.1 Входни и изходни итератори

Реализация на зареждане на контейнер от файл с итератори и **алгоритъм copy и back_inserter**:

Функцията back_inserter връща back_insert_iterator от обекта. Това е итераторен адаптер, осигуряващ :

- Запис на копираната информация;
- Увеличаване на итератора след записа.

```
vector<int> v;
ifstream ifile ("example_file");
copy (istream_iterator<int> (ifile),      // от файлов поток
          istream_iterator<int> (),       // край на файлов поток
          back_inserter(v) );
```

# 4.2.2 Нарастващи итератори

Добавени член функции към входни и изходни итератори:

- Подразбиращ се конструктор
- Оператор за присвояване
- Поведението на оператора за дереферииране (инкрементиране) се променя:

***Многопасов, т.е. дереферирането и инкрементирането не са свързани***

# 4.2.2 Нарастващи итератори

Пример за използване – линейно търсене на обект в итерационен интервал:

```
template<class ForwardIterator, class T>
ForwardIterator find_linear (ForwardIterator first,
                             ForwardIterator last, T& value) {
  while (first != last)
   if (*first == value)
      return first;
    else first++;
  return last;
}
```

# 4.2.2 Нарастващи итератори

Използване на алгоритъма:

```
vector<int> v (3, 1);
v.push_back (7);       // вектор v: 1 1 1 7
vector<int>::iterator it = find_linear (vect.begin(),
                                        vect.end(), 7);
if (it != vect.end() ) cout << *it; else cout << "not found";
```

Изход: 7

# 4.2.3 Двупосочни итератори

Допълнение към двупосочните итератори (допълнително към нарастващите):

**Оператор за намаляване (--)**

Пример – достъп в обратен ред:

```
list<int> lst (1, 1);
lst.push_back (2); // Съдържание : 1 2
list<int>::iterator first = lst.begin();
list<int>::iterator last  = lst.end();
while (last != first) {
  --last;
  cout << *last << " ";
}
```

Изход:2  1

# 4.2.3 Двупосочни итератори

Пример – алгоритъм за сортиране на обекти:

```
template <class BidirectionalIterator, class Compare>
void bubble_sort (BidirectionalIterator first,
     BidirectionalIterator last,
                Compare comp) {
BidirectionalIterator left_el = first, right_el = first;
 right_el++;
 while (first != last)  {
    while (right_el != last) {
      if (comp(*right_el, *left_el))
         iter_swap (left_el, right_el);
      right_el++;
      left_el++;
    }
    last--;
    left_el = first, right_el = first;
    right_el++;
  }
}
```

# 4.2.3 Двупосочни итератори

Пример за сортиране в прав ред:

```
list<int> lst;
ifstream ifile ("example_file.txt");// 10 5 1 4 2 7 3
// fill list from file
copy (istream_iterator<int> (ifile),
            istream_iterator<int> (),
            back_inserter(lst) );
//сортиране в нарастващ ред
bubble_sort (l.begin(), l.end(), less<int>() );
list<int>::iterator first = lst.begin();
list<int>::iterator last  = lst.end();
// тестов изход
while (first != last) {
  cout << *first << " ";
  first++;
}
```

# 4.2.3 Двупосочни итератори

Пример за сортиране в намаляващ ред:

```
bubble_sort (l.begin(), l.end(), greater<int>() );
first = lst.begin();
last  = lst.end();
// тестов изход
cout << endl;
while (first != last) {
  cout << *first << " ";
  first++;
}
```

Изход: 1 2 3 4 5 7 10

10 7 5 4 3 2 1

# 4.2.4 Итератори с произволен достъп

Изисквания към итераторите с произволен достъп (допълнение към двупосочните итератори):

- **оператор+ (int)**
- **оператор += (int)**
- **оператор - (int)**
- **оператор -= (int)**
- **оператор - ( <итератор с произволен достъп> )**
- **оператор [] (int)**
- **оператор < (<итератор с произволен достъп>)**
- **оператор > (<итератор с произволен достъп>)**
- **оператор >= (<итератор с произволен достъп>)**
- **оператор <= (<итератор с произволен достъп>)**

# 4.2.4 Итератори с произволен достъп

```
vector<int> v (1, 1);
// вектор v: 1 2 3 4
v.push_back (2); v.push_back (3); v.push_back (4);
vector<int>::iterator i = v.begin();
vector<int>::iterator j = i + 2; cout << *j << " "; //Изход:      3
i += 3; cout << *i << " "; // Изход:   4
j = i - 1; cout << *j << " "; //Изход:  3
j -= 2;
cout << *j << " "; //Изход: 1
cout << v[1] << endl; //Изход 2
(j < i) ? cout << "j < i" : cout << "not (j < i)"; cout << endl;
//Изход: j < i
(j > i) ? cout << "j > i" : cout << "not (j > i)"; cout << endl;
//Изход: not (i > j)
i = j;
i <= j && j <= i ? cout << "i and j equal" : cout << "i and j not
     equal"; cout << endl;
//Изход: i and j equal
j = v.begin();
i = v.end();
cout << "iterator distance end - begin = size: " << (i - j);
//Изход: iterator distance end - begin = size:  4
```

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

Дефиниция:

Адаптерите (Adaptors) са темплейтни класове, които предоставят интерфейсно съответствие. Наследяване.

- Имат базов клас;
- Имплементират нова функционалност.

Член функциите на базовия клас могат да се променят, разширяват или скриват.

Могат да се комбинират за постигане на нова финкционалност.

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

- Обратни итератори. Обхват:
- Двупосочните;
- Итераторите с произволен достъп.

Те имплементират итератори, които служат за достъп до елементите на контейнера в обратен ред.

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

Пример за създаване и използване на обратен итератор:

```
list<int> list;
// запълване на list с 1 2 3 4
reverse_bidirectional_iterator<list<int>::iterator,
                  list<int>::value_type,
                  list<int>::reference_type,
                  list<int>::difference_type>
           rListIt (list.end());
cout << * rListIt << " "; // извежда последния елемент
rListIt ++; // !! действието на итератора е обратно на
              // увеличението
cout << * rListIt << " ";
rListIt --;   // !! действието на итератора е обратно на
              // намалението
cout << * rListIt;
```

Изход: 4 3 4

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

Пример за използване с алгоритъм, напр. copy

```
list<int> intlist;
// запълване на intlist с 1 2 3 4
// !! копирането е обратно на естествения ред в контейнера
copy(reverse_iterator<int*, int, int&, ptrdiff_t> (intlist.end()),
     reverse_iterator<int*, int, int&, ptrdiff_t> (intlist.begin()),
     ostream_iterator<int> (cout, " ") );
```

Изход:4 3 2 1

Забележка: За използване на reverse_bidirectional_iterator или reverse_iterator се включва <iterator>

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

За всички последователни контейнери (vector, list и deque) се предоставят член функции rbegin и rend, които връщат съответните обратни итератори.

Пример:

```
list<int> intlist;
// запълване на intlist с 1 2 3 4
// !! копирането е в обратен ред
copy (intlist.rbegin(),intlist.rend(), ostream_iterator<int>(cout, " "));
```

Изход:4 3 2 1

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

- Итератори за добавяне.

Дефиниция: Това са итераторни адаптери, които опростяват добавянето в контейнерите. Принципът е, че записваната стойност срещу итератора за добавяне добавя тази стойност в контейнера, за който е създаден. За да се дефинира позицията се предлагат три различни итераторни адаптера за добавяне:

- back_insert_iterator
- front_insert_iterator
- insert_iterator

front_insert_iterator и back_insert_iterator се конструират върху контейнера и добавят елемента в началото и в края на контейнера съответно. back_insert_iterator изисква контейнер за който е дефинирана push_back, а front_insert_iterator съответно изисква push_front:

```
deque<int> d;
back_insert_iterator<deque<int> > bi (d);
front_insert_iterator<deque<int> >  fi (d);
```

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

- Дефиниция на функциите за добавяне в края, началото и във вътрешността на контейнерите:

Функциите back_inserter, front_inserter inserter връщат съответните итератори за добавяне.

Пример:

```
template <class Container>
back_insert_iterator<Container> back_inserter(Container& x) {
    return back_insert_iterator<Container>(x);
}
template <class Container>
front_insert_iterator<Container> front_inserter(Container& x) {
    return front_insert_iterator<Container>(x);
}
template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i) {
    return insert_iterator<Container>(x, Container::iterator(i));
}
```

# 4.2.5 Адаптери на итератори (Iterator Adaptors)

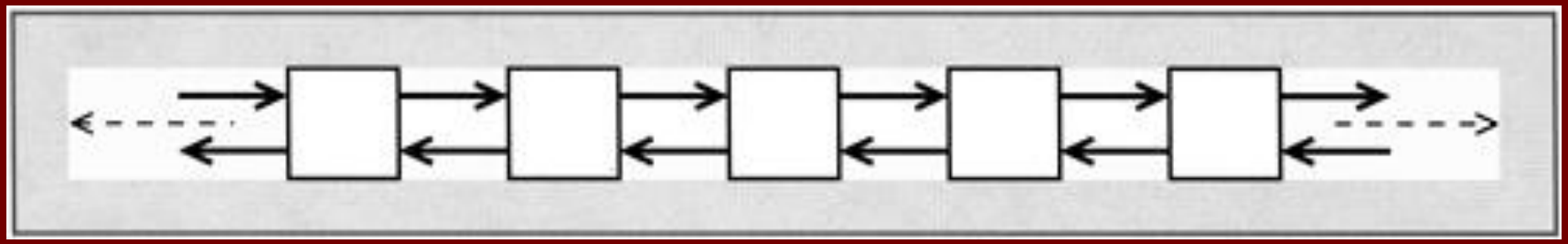
Пример:

```
ifstream f ("example");    // примерен файл: 1 3
deque<int> dObj;
copy (istream_iterator<int, ptrdiff_t>(f),
        istream_iterator<int, ptrdiff_t>(),
        back_inserter(dObj) );
vector<int> vObj (2,7);
copy (vObj.begin(), vObj.end(), front_inserter (dObj) );
insert_iterator<deque<int> > iter = inserter (dObj,
   ++dObj.begin() );
*iter = 9;
```

Изход: 7 9 7 1 3

# 4.3. Списък (list)

Организация:



# 4.3. Списък (list)

## Особености:

- Не осигурява произволен достъп- O(N);
- Добавяне/премахване  е O(1);
- Добавяне/премахване не инвалидизира итератори;
- Обработва операцията без междинни състояния;
- Няма at();
- Няма capacity и realloc. Всеки елемент се създава динамично в собствена памет;
- Списъци предоставят много специални член функции за разместване на елементи-алгоритми (по-бързи), защото те само пренасочват указатели, а не копират и преместват обектите.

# 4.3. Списък (list)

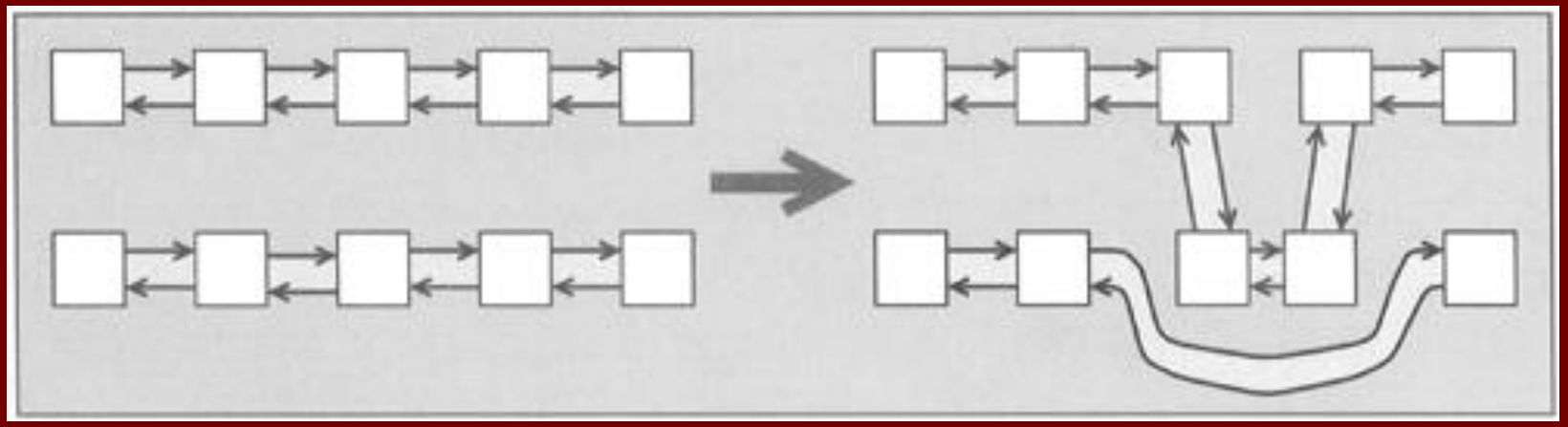
## Член функции (алгоритми):

- remove();
- insert (pos, elem), insert (pos,n, elem), insert (pos, beg,end);
- remove (val), remove_if (op), erase (...);
- resize (...);
- unique(...);
- splice(...);
- sort();
- merge(c2);

- По-бързи от общите алгоритми на STL

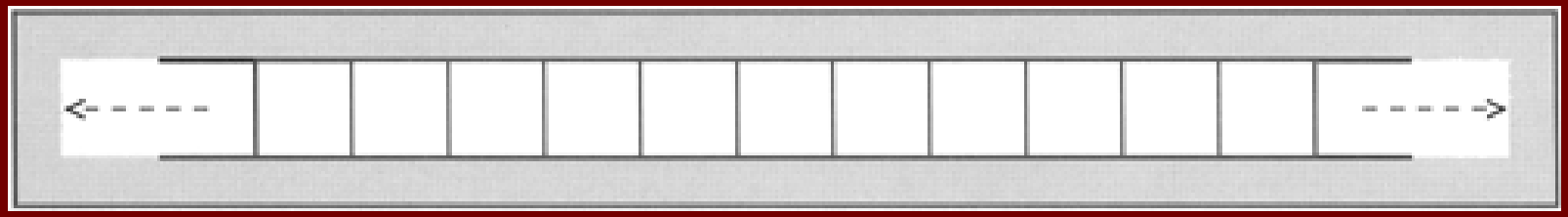
# 4.3. Списък (list)

Преместване на елементи: O(1):



# 4.4. Двойно свързан списък (deque)

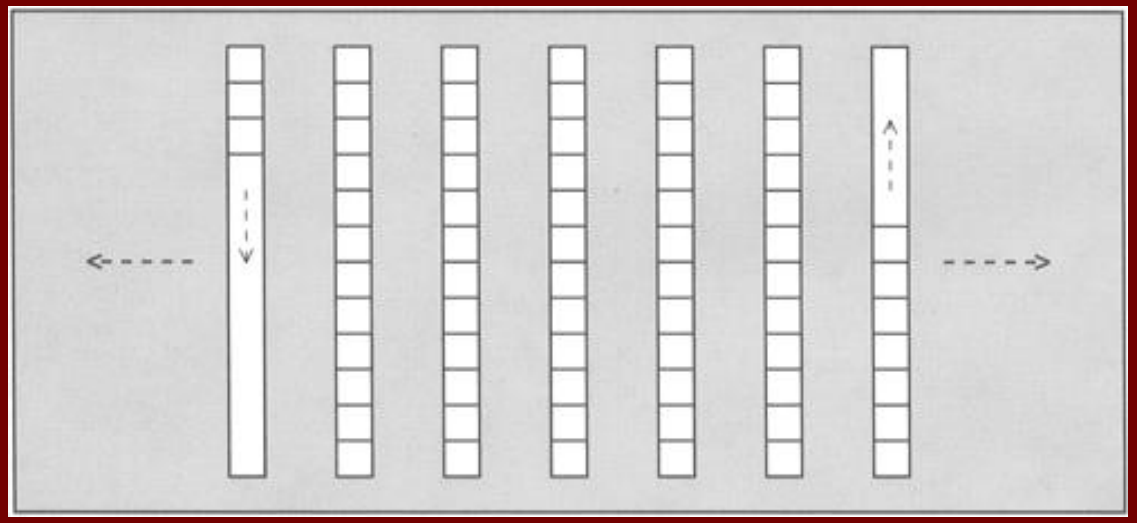
Организация



# 4.4. Двойно свързан списък (deque)

## Особености:

- Добавяне/изтриване на елементи в началото/края - амортизирано константно време;
- Не запазват итераторите при структурни операции, освен в началото или в края;
- По-добро поведение при заемане на нова памет, отколкото вектора. Вътрешна организация:



# 4.4. Двойно свързан списък (deque)

## Особености:

- Интерфейса (публичните функции/оператори) са подобни на вектора;
- Вътрешната структура осигурява два начина за изчислим достъп до елемент, но е малко по-бавно от това за вектора;
- Итераторите трябва да са указатели от специален вид, а не обикновени указатели, защото те трябва да преминават между различните блокове;
- В системи, които имат ограничения в размера на блока памет, deque може да съдържа повече блокове

# Въпроси?