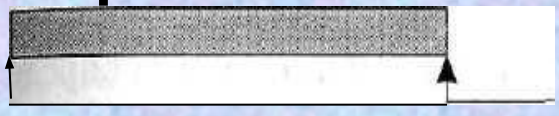
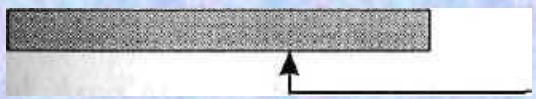
Лекция 2

Операции с последовательными контейнерами

Свойства трех последовательных контейнеров

Для данного типа *T* типы *vector<T>*, *deque<T>* и *list<T>* называются *последовательными контейнерами*. К этой группе относится также массив.

Vector<T> 	**Вставка и удаление здесь. *Произвольный доступ*.**
Deque<T> 	**Вставка и удаление здесь. *Произвольный доступ.***
List<T> 	**Вставка и удаление в любом месте. *Нет произвольного доступа.***

Наборы операций с последов. контейнерами

Операция	Функция	vector	deque	list
Вставить в конце	*push_back*	√	√	√
Удалить в конце	*pop_back*	√	√	√
Вставить в начале	*push_front*	-	√	√
Удалить в начале	*pop_front*	-	√	√
Вставить в любом месте	*insert*	(√)	(√)	√
Удалить в любом месте	*erase*	(√)	(√)	√
Отсортировать	*sort* (алгоритм)	√	√	-

Галочка *(√)* означает, что функции *insert* и *erase* выполняются гораздо медленнее, чем для списков.

Замечания

Говорят, что выполнение функций *insert* и *erase* занимает *линейное время* для векторов и двусторонних очередей, а это означает: время их выполнения пропорционально длине последовательности, хранящейся в контейнере. В противоположность этому

все операции, помеченные галочкой ✓ (без скобок), выполняются за *постоянное время,* то есть время, необходимое для их выполнения, не зависит от длины последовательности.

Рассмотрим программу использования всех функции для вставки и удаления *(push_back, pop_ back, push_front, pop_front, insert* и *erase).*

```
// insdel.cpp:  Вставка и удаление элементов из
// списка.
#include <iostream>
#include <list>
using namespace std;
void showlist(const char *str,  const list<int> &L)
{ // печать содержимого списка
    list<int>::const_iterator  i;
    cout << str << endl <<  "    ";
    for  (i=L.begin(); i != L.end();  ++i)
        cout << *i << " ";  cout  <<  endl;
}
int list_insdel()
{   list<int> L;    int x;
cout << "Enter positive integers, followed by  0:\n";
    while  (cin  >>  x,  x  !=  0)
        L.push_back(x);
showlist("Initial  list:",  L);
```

Посмотрим результат:

Enter positive integers, followed by 0:
10 20 30 0
Initial list:
10 20 30

```
L.push_front(123);
showlist("After inserting 123 at the beginning:", L);
list<int>::iterator i = L.begin();
L.insert(++i, 456);
showlist("After inserting 456 at the second position:", L);
i = L.end();
L.insert(--i,  999);
showlist("After inserting 999 just before the end:",  L);
i = L.begin();
x = *i;
L.pop_front();
cout << "Deleted at the beginning: "  << x << endl;
showlist("After this deletion:",  L);
```

Посмотрим результат:

After inserting 123 at the beginning:
123 10 20 30
After inserting 456 at the second position:
123 456 10 20 30
After inserting 999 just before the end:
123 456 10 20 999 30
Deleted at the beginning: 123
After this deletion:
456 10 20 999 30

```
i = L.end();
x = *--i;
L.pop_back();
cout << "Deleted at the end: " << x << endl;
showlist("After this deletion:",  L);
i = L.begin() ;
x = *++i;
cout << "To be deleted: " << x << endl;
L.erase (i);
showlist("After this deletion (of second
element):", L);
return 0;
}
```

Функции для вставки и удаления здесь применяются к списку, поскольку это единственный контейнерный тип, для которого все эти функции определены и выполняются эффективно (см. вышеприведенную таблицу).

Посмотрим результат:

Deleted at the end: 30
After this deletion:
456 10 20 999
To be deleted: 10
After this deletion (of second element):
456 20 999

Приращение итератора можно выполнять с помощь функции advance (i, n).
Например, advance (i, 3)

Для сортировки списка используется метод sort() самого класса list. Например, L.sort();

Замечания

Рассмотрим употребление *const* в первых двух строчках функции *showlist* :

```
void showlist(const char *str,  const list<int> &L)
{   list<int>::const_iterator  i;    …
```

Добавление приставки *const* к параметрам типа указатель или ссылка, как это сделано выше, является хорошей практикой, если такие параметры *не используются для модификации объектов*, на которые они указывают. Поскольку функция *showlist* не модифицирует ни строку *str*, ни список *L*, отсюда происходят два употребления слова *const* на первой из этих двух строчек.

Замечания

На второй строчке мы должны объявить переменную *i* типа *const_iterator*, чтобы иметь возможность использовать ее вместе с *L*. Это похоже на применение модификатора *const* к указателям: если хотим присвоить вышеобъявленный параметр *str* указателю *p*, мы сможем сделать это, только использовав *const* при объявлении этого указателя:

```
const char *p; // const необходим, поскольку str
p = str;       // описан как   const char *str
               // типа const char *
```

Стирание подпоследовательности

Если [i1,i2) является действительным диапазоном для вектора *v*, мы можем стереть подпоследовательность в *v*, заданную этим диапазоном, следующим образом:

```
v.erase(i1,  i2);
```

То же самое относится и к остальным контейнерам.

Сортировка вектора

```
#include  <iostream>
#include  <vector>
#include  <algorithm>
using namespace std;
int vector_sort1()
{   vector<int>  v;  int  x;
cout << "Enter positive  integers, followed by  0:\n";
while (cin >> x, x !=  0)  v.push_back(x);
    sort (v.begin(),  v.end());
    cout << "After  sorting:  \n" ;
    vector<int>::iterator i;
    for (i=v.begin();i != v.end(); ++i)
        cout << *i << " " ;
    cout << endl;
    return 0;
}
```

Замечания

Вывод этой программы содержит введенные пользователем целые числа, *отсортированные в восходящем порядке.*

Вышеприведенный вызов функции *sort* отличается от вызовов *push back, insert, begin* и др. Поскольку мы пишем не *v.sort(...)*, а просто *sort(...)*, видно, что *sort* является не функцией-членом класса *vector*, а шаблонной функцией, которая не является членом класса.

Технический термин, обозначающий такую шаблонную функцию в STL,- *обобщенный (generic) алгоритм*, или просто *алгоритм.*

Строчка

```
#include <algorithm>
```

необходима, поскольку мы используем алгоритм *sort.*

Сортировка массива

```
#include  <iostream>
#include  <algorithm>
using  namespace  std;
int  sort2()
{    int a[10], x, n = 0, *p;
cout << "Enter  at most  10  positive  integers,  followed by  0:\n";
while (cin >> x, x != 0 && n < 10) a[n++] = x;
    sort(a, a+n);
cout << "After sorting:  \n";
for (p=a; p != a+n; p++) cout << *p << " ";
    cout <<  endl;   return  0;   }
```

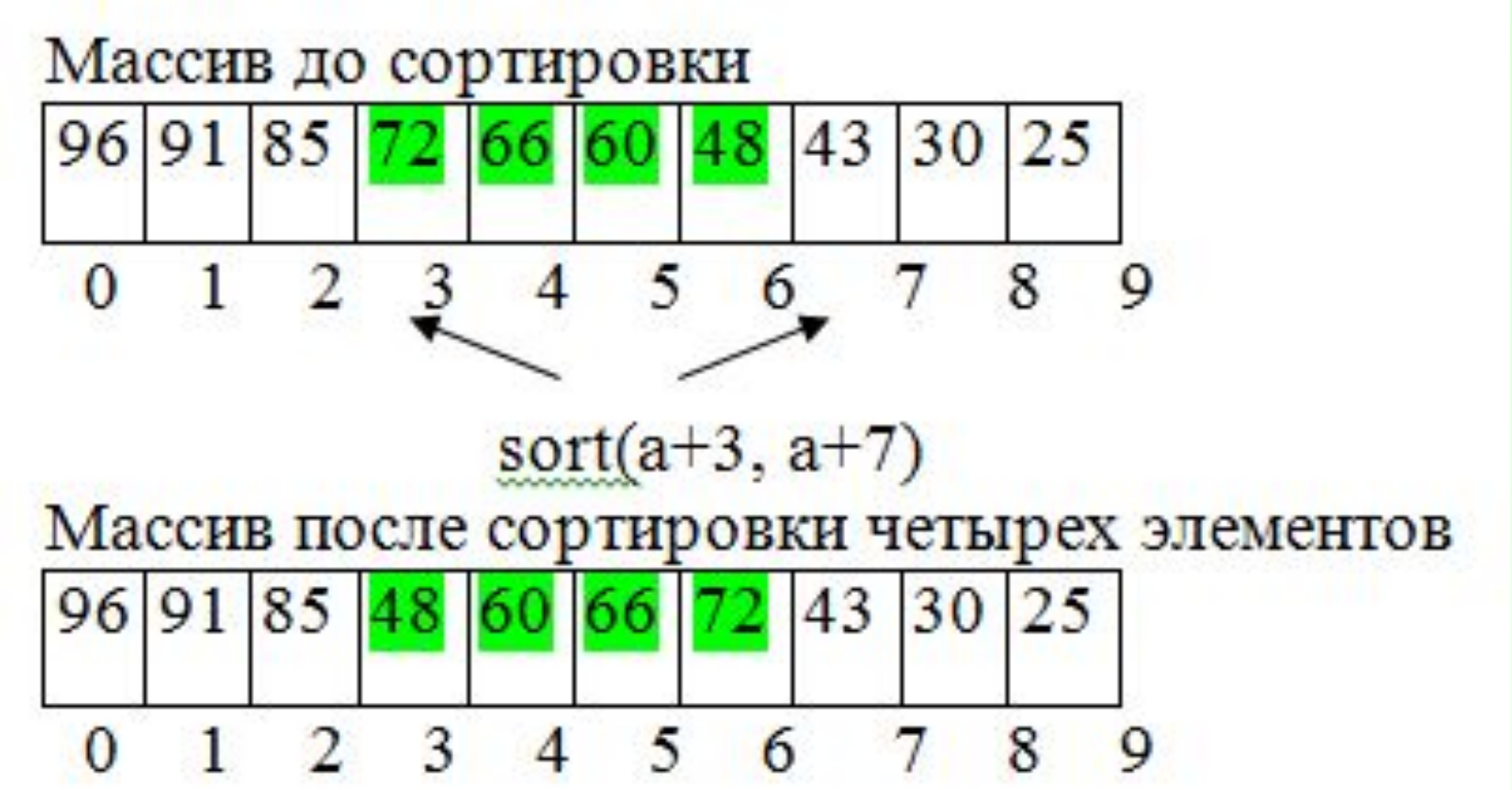
Важно заметить сходство между вызовами:

```
sort(v.begin(), v.end()); // в программе sort1.cpp и
sort(a, a+n); // в программе sort2.cpp.
```

Второй аргумент ссылается на позицию, находящуюся *непосредственно после последнего элемента*.

Сортировка подпоследовательности массива

Например, мы можем отсортировать только элементы *a[3], a[4], a[5]* и *a[6],* написав: sort(a+3, a+7); или, что эквивалентно, sort([&a[3], &a[7]);



Сортировка подпоследовательности вектора

В программе *sort1.cpp* в векторе *v* также отсортируем *v[3], v[4], v[5]* и *v[6]*, написав:

```
vector<int>  v;
vector<int>::iterator i, j;
i = v.begin() + 3;
j = v.begin() + 7;
sort (i, j);    или
sort (v.begin() + 3, v.begin() + 7);
```

Доступ по индексу возможен, т.к. вектор является контейнером *произвольного доступа*, для которого определен *operator[] доступа по индексу*. Но, мы не можем заменить v[3] на *(v + 3), потому что тип переменной v является классом, для которого не определен ни бинарный оператор +, ни унарный оператор *.

Алгоритм STL *sort*

Алгоритм *sort* требует *произвольного доступа.* Такой доступ обеспечивают векторы, массивы и двусторонние очереди, поэтому мы могли использовать этот алгоритм в программах *sort1.cpp* и *sort2.cpp.*

Вызов sort (v.begin () , v.end());

будет работать, если мы в программ заменим всюду *vector* на *deque*, *но не на list.*

Список не обеспечивает произвольного доступа, поэтому с нему не применим алгоритм sort. Для сортировки списка используется метод sort() самого класса list. Например, L.sort();

Инициализация контейнеров

```
int a[3] = {10, 5, 7}; // инициализация массива
int b[ ] = {8, 13}; // эквивалентно int b[2] = {8, 13};
int c[3] = {4}; // эквивалентно int c[3] = {4, 0, 0};
```

Инициализация также возможна и для трех других типов последовательных контейнеров:

```
vector<int> v(a, a+3);   // int a[3] = {10, 5, 7};
deque<int> w(a, a+3);
list<int> x(a, a+3);
```

Но не только массив, а также *vector, deque и list* могут служить основой для инициализации контейнера *того же типа:*

```
vector<int> v1(v.begin(),  v.end());
```

вектор v1 станет идентичен вектору v, оба будут состоять из трех элементов типа int: 10, 5 и 7.

Инициализация контейнеров

Однако, мы не можем использовать значения списка *x* для инициализации вектора *v1*.

```
vector<int>  v1(x.begin(),  x.end()); // не откомпилируется
```

Способ инициализации обеспечивается *конструкторами контейнерных классов.*
Существуют конструкторы, у которых первый параметр указывает размер и второй (необязательный) – значения всех элементов; можно написать:

```
vector<int> v(5,8); // Пять элементов, все = равны 8.
vector<int> v(5);
```

В последнем случае вектор v будет содержать пять элементов, присваивать значения элементам будем позже.

Алгоритм *find*

```
#include <vector>
#include <algorithm>
using namespace std;
int find1()
{   vector<int>  v;  int  x;
cout << "Enter positive integers, followed by 0:\n";
while (cin  >> x, x !=  0)
v.push_back(x);
cout << "Value to be searched for: ";  cin >> x;
vector<int>::iterator  i = find(v.begin() , v.end(), x);
if (i == v.end())  cout << "Not found\n";
else
     {   cout << "Found";
     if (i == v.begin())  cout << " as the first element";
          else cout << " after " << *--i;
      }          // Алгоритм find применим к вектору,
cout  <<  endl;   // двуст-й очереди, списку и массиву
return  0;
}
```

Алгоритм *find* для массива

```
#include <vector>
#include <algorithm>
using namespace std;
int find2()
{
int a[10] , x, n = 0;
cout << "Enter at most 10 integers, followed by  0:\n";
while (cin >> x, x != 0 && n < 10) a[n++]  = x;
cout  <<  "Value  to be searched  for:  ";  cin  >>  x;
int *p = find (a, a+n, x) ;
if  (p == a+n)    cout << "Not found\n";
else
     {   cout  <<  "Found";
     if (p== a)   cout << " as the first element";
          else cout << " after " << *--p;
     }
cout  <<  endl;
return  0;
}
```

Алгоритм *copy*

```
//  copy1.cpp:  Копируем вектор в список.
// Первая версия:  режим замещения.
#include <vector>
#include <list>
int copy_vector_list1()
{
     int a[4] = {10, 20, 30, 40};
     vector<int> v(a, a+4);
     list<int>  L(4);  // Список  из  4  элементов
copy(v.begin(), v.end(), L.begin());
     list<int>::iterator i;
     for (i=L.begin(); i != L.end(); ++i)
          cout << *i << "  ";   // Результат: 10 20 30 40
     cout  <<  endl;
     return  0;
}
```

Алгоритм *copy* и итератор вставки

Рассмотрим режим вставки.

```
list<int> L;       // Пустой список.
```

Заменим вызов алгоритма copy на следующий:

```
copy(v.begin(), v.end(), inserter(L, L.begin()));
```

inserter(…) называется *итератором вставки.*

```
// copy2.cpp:  Вторая версия:  режим вставки.
int copy_vector_list2()
{   int a[4] = {10,  20,  30,  40};
vector<int> v(a,  a+4);
list<int> L(5, 123); // 5 элементов, каждый =123
list<int>:: iterator i  = L.begin();   // ++i;  ++i;
advance (i, 3); // приращение итератора
copy(v.begin(), v.end(), inserter(L, i));
for (i=L.begin(); i != L.end(); ++i)
cout << *i << " ";  cout  <<  endl;   return  0;
}    // Результат: 123 123 10 20 30 40 123 123 123
```

Краткие выводы

1. **Были рассмотрены операции с последовательными контейнерами *(push_back, pop_ back, push_front, pop_front, insert и erase).***
2. **Были рассмотнрены алгоритмы (#include <algorithm>):**

- *сортировки* (векторы, массивы, двуст. очереди), для списка этот алгоритм не применим, используется метод метод sort() самого класса list.

```
sort (v.begin(),  v.end());
sort([&a[3], &a[7]);
```

Но L.sort();

- *поиска* (векторы, массивы, двуст. очереди, списки)

```
i = find(v.begin() , v.end(), x);
int *p = find (a, a+n, x) ;
```

- для массива

Краткие выводы

- *копирования* (векторы, массивы, двуст. Очереди, списки):

1) в режиме замещения

```
copy(v.begin(), v.end(), L.begin());
```

1) в режиме вставки (используется итератор вставки inserter(…))

```
copy(v.begin(), v.end(), inserter(L, i));
```