

Севастопольский государственный университет  
Кафедра «Информационные системы»

Курс лекций по дисциплине  
**«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ  
ПРОГРАММИРОВАНИЕ»**  
(ООП)

Лектора: Пелипас Всеволод Олегович  
Сметанина Татьяна Ивановна

## Лекция 11

КОНТЕЙНЕРЫ  
ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ. ИТЕРАТОРЫ.  
АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ.  
АЛГОРИТМЫ STL.

# Контейнеры

**Контейнер** — это объект, содержащий другие объекты. В него можно добавлять объекты и удалять из него объекты.

Примерами контейнеров могут служить массивы, линейные списки или стеки.

Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнеры, а также алгоритмы и итераторы, называют **стандартной библиотекой шаблонов (STL — Standard Template Library)**.

# Контейнеры

Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.

## Стандартные контейнерные классы STD:

**последовательные** — обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности; (векторы (**vector**), двусторонние очереди (**deque**) и списки (**list**), а также так называемые адаптеры (то есть варианты) контейнеров — стеки (**stack**), очереди (**queue**) и очереди с приоритетами (**priority\_queue**). Каждый вид контейнера обеспечивает свой набор действий над данными.

**ассоциативные** контейнеры обеспечивают быстрый доступ к данным по ключу, построены на основе сбалансированных деревьев (словари (**map**), словари с дубликатами (**multimap**), множества (**set**), множества с дубликатами (**multiset**) и битовые множества (**bitset**)).

# Последовательные контейнеры

- **vector** — динамический массив — структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.
- **deque** — двусторонняя очередь (дек) — эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов
- **list** — линейный список — эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам
- **stack** — стек,
- **queue** — очередь,
- **priority\_queue** — очередь с приоритетами;

# vector

Объект, хранимый в **vector**, должен иметь:

- конструктор по умолчанию;
- операторы **<** и **==**.

Возможности **vector**:

В классе вектор определены операторы сравнения **==**, **<**, **<=**, **!=**, **>**, **>=**.

Новые элементы могут включаться с помощью функций **insert()**, **push\_back()**, **resize()**, **assign()**.

Существующие элементы могут удаляться с помощью функций **erase()**, **pop\_back()**, **resize()**, **clear()**.

Доступ к отдельным элементам осуществляется с помощью итераторов **begin()**, **end()**, **rbegin()**, **rend()**.

Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций, подключаемых заголовочным файлом **<algorithm>**

# Примеры вызова конструкторов вектора

а) `vector <one> m1 (10);`

`/* вектор из 10 объектов класса one (работает конструктор one без параметров) */`

б) `vector <int> v1 (10, 1);` // вектор из 10 единичных элементов

в) `vector <int> v2 (v1.begin(), v1.begin() + 1);` `/* вектор из двух элементов, равных первым двум элементам v1 */`

г) `vector <int> v3 (v1);` // вектор, равный вектору v1

д) `vector <Wword> m2 (5, Wword("test"));` `/* вектор из 5 объектов класса Wword заданным именем (работает конструктор с параметром char*) */`

# Основные типы, используемые в шаблонах

<b>value_type</b>	- тип элемента контейнера;
<b>size_type</b>	- тип индексов элементов и т. д.;
<b>iterator</b>	- тип "итератор" – указатель на элемент;
<b>const_iterator</b>	- тип "константный итератор" – используется для неизменных данных;
<b>reverse_iterator</b>	- обратный итератор – для просмотра от конца к началу;
<b>const_reverse_iterator</b>	- константный обратный итератор;
<b>key_type</b>	- тип ключа (для ассоциативных контейнеров);
<b>key_compare</b>	- тип результата сравнения ключей.
<b>reference</b>	- ссылка на элемент;
<b>const_reference</b>	- константная ссылка на элемент;



# Итераторы

**Итераторы (iterators)** – это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера и сканировать его элементы, примерно так же, как указатели используются для доступа к элементам массива. С итераторами можно работать так же, как с указателями. К ним можно применять операции `*`, инкремент, декремент. Тип итератора **iterator** определён в различных контейнерах.

Существует пять типов итераторов:

1. **Итераторы ввода (input iterator)** поддерживают операции равенства, разыменования и инкремента: `==`, `!=`, `*i`, `++i`, `i++`, `*i++`. Специальным случаем итератора ввода является **istream\_iterator**.

2. **Итераторы вывода (output iterator)** поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: `++i`, `i++`, `*i = t`, `*i++ = t`. Специальным случаем итератора вывода является **ostream\_iterator**.

# Итераторы

3. **Однонаправленные итераторы (forward iterator)** поддерживают все операции итераторов ввода/вывода и позволяют без ограничения применять присваивание:  $==$ ,  $!=$ ,  $=$ ,  $*i$ ,  $++i$ ,  $i++$ ,  $*i$ .

4. **Двунаправленные итераторы (bidirectional iterator)** обладают всеми свойствами **forward-итераторов**, а также имеют дополнительную операцию декремента ( $--i$ ,  $i--$ ,  $*i--$ ), что позволяет им проходить контейнер в обоих направлениях.

5. **Итераторы произвольного доступа (random access iterator)** обладают всеми свойствами **bidirectional-итераторов**, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ к элементу по индексу:  $i += n$ ,  $i + n$ ,  $i -= n$ ,  $i - n$ ,  $i1 - i2$ ,  $i[n]$ ,  $i1 < i2$ ,  $i1 \leq i2$ ,  $i1 > i2$ ,  $i1 \geq i2$ .

В STL также поддерживаются обратные итераторы (**reverse iterators**). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, проходящие последовательность в обратном направлении.

# Операции над итераторами

Категория итератора	Допустимые операции	Контейнеры
Входной ( <b>input</b> ) – перемещается вперед и допускает только чтение элементов	$x = *i,$ $++i, i++$	все
Выходной ( <b>output</b> ) – перемещается вперед и допускает запись элементов	$*i = x,$ $++i, i++$	все
Прямой ( <b>forward</b> ) – допускает запись и чтение элементов при движении вперед. Реверсивный ( <b>reverse</b> ) – допускает запись и чтение элементов при движении назад.	$x = *i, *i = x,$ $++i, i++$ $x = *i, *i = x,$ $--i, i--$	все
Двунаправленный ( <b>bidirectional</b> ) – работает вперед и назад	$x = *i,$ $++i, i++, --i, i--$	все
Произвольного доступа ( <b>random access</b> )	$x = *i, *i = x,$ $--i, i--$ $i + n, i - n,$ $i += n, i -= n$ $i < j, i > j,$ $i <= j, i >= j$	все, кроме list

# Методы получения адресов для инициализации итераторов

Указатель на первый:

**iterator begin()**

**const\_iterator begin() const**

Указатель на следующий за последним:

**iterator end()**

**const\_iterator end() const**

Указатель на первый при обратном просмотре:

**reverse\_iterator rbegin()**

**const\_reverse\_iterator rbegin() const**

Указатель на элемент, следующий за последним, при обратном просмотре:

**reverse\_iterator rend()**

**const\_reverse\_iterator rend() const**

# Методы доступа к элементам контейнеров STL

Метод	Описание
<b>front()</b>	ссылка на первый элемент
<b>back()</b>	ссылка на последний элемент
<b>operator [](i)</b>	доступ по индексу без проверки
<b>at(i)</b>	доступ по индексу с проверкой

# Методы для включения и исключения элементов

Операция	Метод	vector	deque	list
Вставка в начало	<b>push_front()</b>	-	+	+
Удаление из начала	<b>pop_front()</b>	-	+	+
Вставка в конец	<b>push_back()</b>	+	+	+
Удаление из конца	<b>pop_back()</b>	+	+	+
Вставка в произвольное место	<b>insert()</b>	+	+	+
Удаление из произвольного места	<b>erase()</b>	+	+	+
Произвольный доступ	<b>[ ], at</b>	+	+	-

# Другие операции с контейнерами STL

**size()** - количество элементов

**empty()** - определяет, пуст ли контейнер

**capacity()** - память, выделенная под вектор (только для векторов)

**reserve(n)** - выделяет память для контейнера под **n** элементов

**resize(n)** - изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)

**swap(x)** - обмен местами двух контейнеров

**==, !=, <** - операции сравнения

**operator =(x)** - контейнеру присваиваются элементы контейнера **x**

**assign(n, x)** - присваивание контейнеру **n** копий элементов **x** (не для ассоциативных контейнеров)

**assign(first, last)** - присваивание элементов из диапазона **[first:last]**

**operator [](k)** - доступ к элементу с ключом **k**

**find(k)** - находит элемент с ключом **k**

**lower\_bound(k)** - находит первый элемент с ключом, меньшим **k**

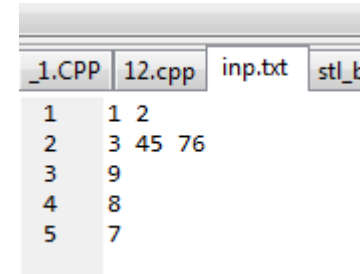
**upper\_bound(k)** - находит первый элемент с ключом, большим **k**

**equal\_range(k)** - находит **lower\_bound** (нижнюю границу) и **upper\_bound** (верхнюю границу) элементов с ключом **k**

# Пример использования

```
#include <fstream>
#include <vector>
using namespace std;
void main()
{
    ifstream in ("inp.txt");
    vector<int> v; // инстанцирование шаблона

    int x;
    while ( in >> x )
        v.push_back(x); // добавление элемента в конец
    for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
        cout << *i << " ";
    system("pause");
}
```



A screenshot of a text editor window with four tabs: "\_1.CPP", "12.cpp", "inp.txt", and "stl\_b". The "inp.txt" tab is active, displaying the following content:

1	1 2
2	3 45 76
3	9
4	8
5	7



A screenshot of a terminal window showing the output of the program. The output is a single line of numbers: "1 2 3 45 76 9 8 7".



# Пример использования

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
    unsigned limit = 1000;
    vector <unsigned> numbers;
    numbers.reserve(limit - 1);
    for (unsigned i = 2; i < limit; ++i)
        numbers.push_back(i);
    for (size_t i = 0; i < numbers.size(); ++i) {
        size_t j = i + 1;
        while (j < numbers.size()) {
            if (numbers[j] % numbers[i] == 0)
                numbers.erase((vector<unsigned>::iterator)&numbers[j]);
            else
                ++j;
        }
    }
}
```

# Пример использования

```
for (vector<unsigned>::iterator i = numbers.begin();  
      i != numbers.end(); ++i)
```

```
    cout << *i << " ";
```

```
numbers.clear();
```

```
}
```

```
// обработка 2 вариант
```

```
for (vector<unsigned>::iterator i = numbers.begin(); i !=  
      numbers.end(); ++i)
```

```
{  
    vector<unsigned>::iterator j = i + 1;
```

```
    while (j != numbers.end()) {
```

```
        if (*j % *i == 0)
```

```
            numbers.erase(j);
```

```
        else
```

```
            ++j;
```

```
    }
```

```
}
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997
```

# Ассоциативные контейнеры

- **map** — словарь уникальных ключей,
- **multimap** — словарь ключей с дубликатами,
- **set** - множество,
- **multiset** — мультимножество,
- **bitset** — битовое множество (набор битов).

Словари часто называют также *ассоциативными массивами* или *отображениями*.

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент.

**map** - это последовательность пар {ключ, значение}, которая обеспечивает быстрое получение значения по ключу. Контейнер **map** предоставляет двунаправленные итераторы.

Контейнер **map** требует, чтобы для типов ключа существовала операция  $<$ . Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Компонентная функция **find()** ищет элемент контейнера по заданному ключу.

# Пример работы с ассоциативным контейнером

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
typedef map <string, long, less <string> > map_sl;
int main()
{   map_sl m;   string str;   long num;
    ifstream in("inp.txt");
    while (in >> num)
    {
        in.get();
        getline(in, str);
        m[str] = num;
        cout << str << " " << num << endl;
    }
}
```

# Пример работы с ассоциативным контейнером

// Вывод элемента по ключу

```
cout << "Name: " << endl;
```

```
getline(cin, str); // ВВОД КЛЮЧА
```

```
if (m.find(str) != m.end()) // ПОИСК ЗНАЧЕНИЯ ПО КЛЮЧУ
```

```
    cout << m[str]<<endl;
```

```
else
```

```
    cout << "Error" ;
```

```
system("pause");
```

```
}
```

	_1.CPP	[*] 12.cpp	inp.txt	stl_
1	1	uuu		
2	2	aaa		
3	3	ttt		
4	4	ooo		
5	5	ppp		

```
uuu 1
aaa 2
ttt 3
ooo 4
ppp 5
Name:
aaa
2
n
```

# Работа с ассоциативным массивом map

```
#include <iostream>
#include <string>
#include <map> //подключили библиотеку для работы с map
using namespace std;
int main() {
    //Сделаем и инициализируем map
    map <int,string> myFirstMap;
    myFirstMap.insert(pair<int, string>(17, "apple"));
    myFirstMap.insert(pair<int, string>(40, "milk"));
    myFirstMap.insert(pair<int, string>(17, "bread")); //не вставится!
    myFirstMap.insert( pair<int, string>(20, "fish"));
    for (map <int,string>::iterator it = myFirstMap.begin(); it !=
        myFirstMap.end(); it++)
        cout << it->first << " : " << it->second << endl;
```

# Работа с ассоциативным массивом map

//Ещё один map

char c='a';

map <char, int> mySecondMap;

for (int i = 0; i < 5; ++i,++c)

mySecondMap.insert ( pair<char, int>(c,i+1));

mySecondMap.insert ( pair<char, int>('d',5) ); //не вставится!

mySecondMap.insert(mySecondMap.begin(), make\_pair('e',1));

//так тоже не вставится

mySecondMap.insert(mySecondMap.begin(), make\_pair('z',0) );

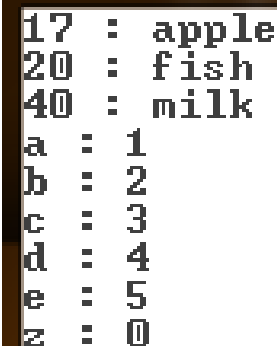
//а так можно

for (map <char, int>::iterator it = mySecondMap.begin(); it !=  
mySecondMap.end(); it++)

cout << (\*it).first << " : " << (\*it).second << endl;

cin.get(); return 0;

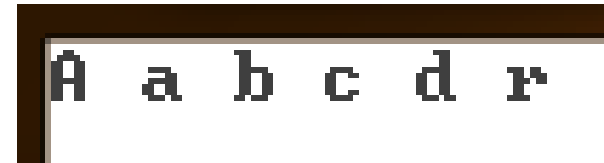
}



```
17 : apple
20 : fish
40 : milk
a : 1
b : 2
c : 3
d : 4
e : 5
z : 0
```

# Работа с множеством set

```
#include <iostream>
#include <string>
#include <set> //заголовочный файл множеств и мультимножеств
#include <iterator>
using namespace std;
int main() {
set <char> mySet;
string str("Abracadabra");
for (int i=0; i<str.length(); i++) mySet.insert(str[i]);
copy(mySet.begin(), mySet.end(), ostream_iterator<char>(cout, " "));
cin.get(); return 0;
}
```



A a b c d r



# Алгоритмы STL

*Алгоритмы (algorithms)* выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, просмотра, сортировки, поиска, замены содержимого контейнеров.

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, один и тот же алгоритм может работать с разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе.

Алгоритмы определены в заголовочном файле `<algorithm>`, приведём имена некоторых наиболее распространённых функций-алгоритмов.

# Алгоритмы STL

Алгоритм	Назначение
<code>accumulate</code>	Вычисление суммы элементов в заданном диапазоне
<code>copy</code>	Копирование последовательности, начиная с первого элемента
<code>count</code>	Подсчет количества вхождений значения в последовательность
<code>count_if</code>	Подсчет количества выполнений условия в последовательности
<code>equal</code>	Попарное равенство элементов двух последовательностей
<code>fill</code>	Замена всех элементов заданным значением
<code>find</code>	Нахождение первого вхождения значения в последовательность
<code>find_first_of</code>	Нахождение первого значения из одной последовательности в другой
<code>find_if</code>	Нахождение первого соответствия условию в последовательности
<code>for_each</code>	Вызов функции для каждого элемента последовательности
<code>merge</code>	Слияние отсортированных последовательностей
<code>remove</code>	Перемещение элементов с заданным значением
<code>replace</code>	Замена элементов с заданным значением
<code>search</code>	Нахождение первого вхождения в первую последовательность второй последовательности
<code>sort</code>	Сортировка
<code>swap</code>	Обмен двух элементов
<code>transform</code>	Выполнение заданной операции над каждым элементом последовательности

# Пример

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void myfunction (int i) { cout << ' ' << i; }
int main () {
    vector<int> myvector;
    myvector.push_back(10);
    myvector.push_back(20);
    myvector.push_back(30);
    for_each (myvector.begin(), myvector.end(), myfunction);
    cin.get(); return 0;
}
```

---

10 20 30