

# Стандартная библиотека шаблонов STL

STL (Standard Template Library) является частью стандарта C++.

**Ядро STL** состоит из четырех основных компонентов:

**контейнеры,**

**итераторы,**

**алгоритмы,**

**распределители памяти.**

# Контейнеры

**Контейнер** – это класс, который предназначен для хранения объектов какого-либо типа.

Примеры известных ранее контейнеров:

- таблица идентификаторов,
- массив,
- дерево,
- список,
- ассоциативный список, например, список, хранящий фамилии людей и номера их телефонов, ключом которого является фамилия, если она уникальна(!).

# Стандартные контейнеры STL

- Vector < T >** - динамический массив
- List < T >** - линейный список
- Stack < T >** - стек
- Queue < T >** - очередь
- Deque < T >** - двусторонняя очередь
- Priority\_queue < T >** - очередь с приоритетами
- Set < T >** - множество
- Bitset < N >** - множество битов (массив из N бит)
- Multiset < T >** - набор элементов, возможно, одинаковых
- Map < key, val >** - ассоциативный список
- Multimap < key, val >** - ассоциативный список для хранения пар ключ/значение, где с каждым ключом может быть связано более одного значения.

# Состав контейнеров

В каждом классе-контейнере определен набор методов для работы с контейнером, причем все контейнеры поддерживают **стандартный набор базовых операций**.

**Базовая операция контейнера (базовый метод)** – метод класса, имеющий во всех контейнерах одинаковое имя, одинаковый прототип и семантику (их примерно 15-20).

Например,

функция **push\_back ()** помещает элемент в конец контейнера,  
функция **size ()** выдает текущий размер контейнера.

Базовыми методами можно пользоваться одинаково независимо от того, в каком конкретно контейнере находятся элементы, можно также менять контейнеры, не меняя тела функций, работающих с ними.

Операции, которые не могут быть эффективно реализованы для всех контейнеров, не включаются в набор общих операций.

Например, обращение по индексу введено для контейнера **vector** , но не для **list**.

# Типы, используемые в контейнерах

Каждый контейнер в своей **public** части содержит серию **typedef**, где введены стандартные имена типов, например:

<b>value_type</b>	- тип элемента,
<b>allocator_type</b>	- тип распределителя памяти,
<b>size_type</b>	- тип, используемый для индексации,
<b>iterator</b>	- итератор,
<b>const_iterator</b>	- константный итератор,
<b>reverse_iterator</b>	- обратный итератор,
<b>const_reverse_iterator</b>	- обратный константный итератор,
<b>pointer</b>	- указатель на элемент,
<b>const_pointer</b>	- указатель на константный элемент,
<b>reference</b>	- ссылка на элемент,
<b>const_reference</b>	- ссылка на константный элемент.

Эти имена определяются внутри каждого контейнера так, как это необходимо, т.е. скрывают реальные типы, что, например, позволяет писать программы с использованием контейнеров, ничего не зная о реальных типах.

В частности, можно составить код, который будет работать с любым контейнером.

# Распределители памяти

Каждый контейнер имеет **распределитель памяти (allocator)**, который используется при выделении памяти под элементы контейнера и предназначен для того, чтобы освободить пользователей контейнеров, от подробностей физической организации памяти.

Стандартная библиотека обеспечивает стандартный распределитель памяти, заданный стандартным шаблоном ***allocator*** из заголовочного файла **<memory>**, который выделяет память при помощи операции **new** (**()**) и по умолчанию используется всеми стандартными контейнерами.

Класс **allocator** обеспечивает стандартные способы выделения и перераспределения памяти, а также стандартные имена типов для указателей и ссылок.

Пользователь может задать свои распределители памяти, предоставляющие альтернативный доступ к памяти.

Стандартные контейнеры и алгоритмы получают память и обращаются к ней через средства, обеспечиваемые распределителем памяти.

# Класс allocator

```
template <class T> class allocator {
public:
    typedef T * pointer;
    typedef T & reference;
    .....
    allocator ( ) throw ( );
    .....
    pointer allocate ( size_type n );    // выделяет память для
                                         // n объектов типа T
    void deallocate ( pointer p, size_type n ); // освобождает память,
                                                // отведенную под n объектов типа T
    void construct ( pointer p, const T & val );
                                         // инициализирует *p значением val
    void destroy ( pointer p );    // вызывает деструктор для *p,
                                   // память при этом не освобождается.
    .....
};
```

# Итераторы

**Итератор** – это класс, объекты которого выполняют такую же роль по отношению к контейнеру, как указатели по отношению к массиву. Указатель может использоваться в качестве средства доступа к элементам массива, а итератор - в качестве средства доступа к элементам контейнера.

Итераторы «склеивают» ядро STL в одну библиотеку.

Итераторы поддерживают абстрактную модель данных как последовательности объектов.

Понятия «нулевой итератор» не существует, а при организации циклов происходит сравнение с концом последовательности.

Каждый контейнер обеспечивает свои итераторы, поддерживающие стандартный набор итерационных операций со стандартными именами и смыслом.

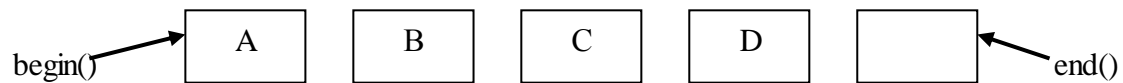
Итераторные классы и функции находятся в заголовочном файле **< iterator >**.



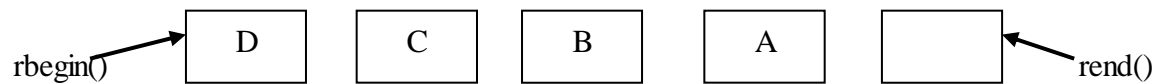
# Методы контейнеров для нахождения значений итераторов концов последовательности элементов

- **iterator begin ( );** – возвращает итератор, который указывает на первый элемент последовательности.
- **const\_iterator begin ( ) const;**
- **iterator end ( );** – возвращает итератор, который указывает на элемент, следующий за последним элементом последовательности.
- **const\_iterator end ( ) const;**
- **reverse\_iterator rbegin ( );** - возвращает итератор, указывающий на первый элемент в обратной последовательности.
- **const\_reverse\_iterator rbegin ( ) const;**
- **reverse\_iterator rend ( );** - возвращает итератор, указывающий на элемент, следующий за последним в обратной последовательности.
- **const\_reverse\_iterator rend ( ) const;**

# Прямые итераторы



# Обратные итераторы



# Операции над итераторами

Пусть  $p$  - объект типа итератор.

К каждому итератору можно применить, как минимум, три ключевые операции:

- $*p$  – элемент, на который указывает итератор,
- $p++$  - переход к следующему элементу последовательности,
- $==$  - операция сравнения.

## Пример:

`iterator p = v.begin();` – такое присваивание верно независимо от того, какой контейнер  $v$  .

Теперь  $*p$  – первый элемент контейнера  $v$ .

## Замечание:

при проходе последовательности как прямым, так и обратным итератором переход к следующему элементу будет  $p++$  (а не  $p--$  !).

**Не все** виды итераторов поддерживают один и тот же набор операций.

# Категории итераторов

В библиотеке STL введено **5** категорий итераторов:

- 1. **Вывода** (output - запись в контейнер)  
( \*p = , ++ )
- 2. **Ввода** (input - считывание из контейнера)  
( = \*p, →, ++, ==, != )
- 3. **Однонаправленный** (forward)  
( \*p =, =\*p, →, ++ , ==, != )
- 4. **Двунаправленный** (bidirectional)  
( \*p=, =\*p, →, ++,--, ==, != ) - list, map, set
- 5. **С произвольным доступом** (random\_access)  
(\*p=, =\*p, →, ++,--, ==, !=, [ ], +, -, +=, -=, <, >, <=, >= ) - vector, deque

Каждая последующая категория является более мощной, чем предыдущая.

# Алгоритмы

**Алгоритмы STL** (их всего 60) - реализуют некоторые распространенные операции с контейнерами, которые не представлены функциями-членами каждого из контейнеров (например, просмотр, сортировка, поиск, удаление элементов...).

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций.

Операции, реализуемые алгоритмами, являются универсальными для любого из контейнеров и поэтому определены вне этих контейнеров.

Реализация алгоритмов не использует имен никаких конкретных контейнеров, а все действия над контейнером производятся через универсальные имена итераторов.

Зная, как устроены алгоритмы, можно писать свои собственные алгоритмы обработки, которые не будут зависеть от контейнера.

Все стандартные алгоритмы находятся в пространстве имен `std`, а их объявления - в заголовочном файле **< algorithm >** .

# Группы алгоритмов

**1. Немодифицирующие алгоритмы** - извлекают информацию из контейнера, но не модифицируют сам контейнер (ни элементы, ни порядок их расположения ).

Примеры :

**Find ()** – находит первое вхождение элемента с заданным значением

**Count ()** – количество вхождений элемента с заданным значением

**For\_each ()** – применяется некоторая операция к каждому элементу  
(не связано с изменением )

**2. Модифицирующие алгоритмы** - изменяют содержимое контейнера. Либо сами элементы меняются, либо их порядок, либо их количество.

Примеры :

**Transform ()** – применяется некоторая операция к каждому элементу  
( каждый элемент изменяется)

**Reverse ()** – переставляет элементы в последовательности

**Copy ()** – создает новый контейнер

**3. Сортировка.**

Примеры :

**Sort ()** – простая сортировка .

**Stable\_sort ()** – сохраняет порядок следования одинаковых элементов  
(например, это бывает существенно при сортировке по нескольким ключам).

**Merge ()** – склеивает две отсортированные последовательности.

# Категории итераторов и алгоритмы

По соглашению, при описании алгоритмов, входящих в STL, используются стандартные имена формальных параметров-итераторов.

В зависимости от названия итератора в прототипе алгоритма, должен использоваться итератор уровня «не ниже чем». То есть по названию параметров шаблона можно понять, какого рода итератор нам нужен, то есть к какому контейнеру применим этот алгоритм.

**Пример:** шаблонная функция **find()** с тремя параметрами ( итератор, с которого начинается поиск, каким заканчивается и искомый элемент).

Для реализации целей функции достаточно итератора ввода (из контейнера).

```
Template < class InputIterator, class T >  
InputIterator find ( InputIterator first, InputIterator last, const T& value ) {  
    while ( first != last && * first != value )  
        first ++;  
    return first;  
}
```

Однако категория итераторов не принимает участия в вычислениях. Этот механизм относится исключительно к компиляции.

# typename

1. Ключевое слово **typename** используется при описании параметра-типа шаблона (наряду с ключевым словом **class**). Например,

```
template <typename T>  
void f (T a) {...}
```

2. Если используемое в шаблоне имя типа зависит от параметров шаблона, **необходимо** использовать ключевое слово **typename**. Например,

```
template <class T>  
void f (vector <T> & v)  
{  
    vector <T> :: iterator i = begin ();           // Err!  
    typename vector <T> :: iterator i = begin(); //O.K.  
    ...  
}
```



# Пример шаблонной функции, использующей тип, вложенный в класс–параметр шаблона

```
struct X {  
    enum { e1, e2, e3 } g;  
    struct inner {  
        int i, j;  
        void g ( ) { cout << "ggg\n"; }  
    };  
    inner c;  
};  
  
template < class T >  
void f ( typename T::inner t ) { t.g ( ); }  
  
int main ( ) {  
    X x;  
    x.g = X::e1;  
    x.c.g ( );  
    X::inner iii;  
    iii.i = 7;  
    f <X> ( iii );  
    return 0;    }
```

# Пример шаблонной функции для контейнеров STL

Поиск заданного элемента в контейнере, начиная с последнего (просмотр контейнера от конца к началу) обычно производится так:

```
template < class C >
typename C :: const_iterator find_last
    ( const C & c, typename C :: value_type v )
{
    typename C :: const_iterator p = c.end ( );
    while ( p != c.begin ( ) )
        if ( * -- p == v )
            return p;
    return c.end ( );
}
```

# Контейнер vector

```
template < class T , class A = allocator < T > > class vector {  
.....  
public:  
// Типы – typedef ..... - см. выше  
// Итераторы ..... - см. выше  
//  
// Доступ к элементам  
//  
reference operator [ ] (size_type n); // доступ без проверки диапазона  
const_reference operator [ ] (size_type n) const;  
  
reference at (size_type n); // доступ с проверкой диапазона (если индекс  
// выходит за пределы диапазона, возбуждается исключение out_of_range)  
const_reference at (size_type n) const;  
  
reference front ( ); // первый элемент вектора  
const_reference front ( ) const;  
  
reference back (); // последний элемент вектора  
const_reference back ( ) const;
```

# Контейнер vector

```
// Конструкторы, деструктор, operator=
```

```
//
```

```
explicit vector (const A&=A()); //создается вектор нулевой длины
```

```
explicit vector (size_type n; const T& value = T(); const A& = A());
```

```
// создается вектор из n элементов со значением value
```

```
// (или с "нулями" типа T, если второй параметр отсутствует;
```

```
// в этом случае конструктор умолчания в классе T обязателен)
```

```
template <class I> vector (I first, I last, const A& = A());
```

```
// инициализация вектора копированием элементов из [first, last),
```

```
// I - итератор для чтения
```

```
vector (const vector < T, A > & obj); // конструктор копирования
```

```
vector& operator = (const vector < T, A > & obj);
```

```
~vector();
```

# Контейнер vector

**//Некоторые функции-члены класса vector**

**//**

**iterator erase (iterator i ); // удаляет элемент, на который указывает данный  
// итератор. Возвращает итератор элемента, следующего за удаленным.**

**iterator erase (iterator st, iterator fin); // удалению подлежат все элементы  
// между st и fin, но fin не удаляется. Возвращает fin.**

**Iterator insert ( iterator i , const T& value = T()); // вставка некоторого  
// значения value перед i. Возвращает итератор вставленного элемента).**

**void insert (iterator i , size\_type n, const T&value); // вставка n копий  
// элементов со значением value перед i.**

**void push\_back ( const T&value ) ; // добавляет элемент в конец вектора**

**void pop\_back () ; // удаляет последний элемент (не возвращает значение!)**

**size\_type size() const; // выдает количество элементов вектора**

**bool empty () const; // возвращает истину, если вызывающий вектор пуст**

**void clear(); //удаляет все элементы вектора**                    **....        }**

# Контейнер list

```
template < class T , class A = allocator < T > > class list {  
.....  
public:  
// Типы  
// .....  
// Итераторы  
// .....  
//  
// Доступ к элементам  
//  
reference front (); // первый элемент списка  
  
const_reference front () const;  
  
reference back (); // последний элемент списка  
  
const_reference back () const;
```

# Контейнер list

```
// Конструкторы, деструктор, operator=
```

```
//
```

```
explicit list (const A& = A()); // создается список нулевой длины
```

```
explicit list (size_type n; const T& value = T(); const A& = A());
```

```
// создается список из n элементов со значением value
```

```
// (или с "нулями" типа T, если второй параметр отсутствует)
```

```
template <class I> list (I first, I last, const A& = A());
```

```
// инициализация списка копированием элементов из [first, last),
```

```
// I - итератор для чтения
```

```
list (const list < T, A > & obj); // конструктор копирования
```

```
list& operator = (const list < T, A > & obj);
```

```
~list();
```

# Контейнер list

**//Некоторые функции-члены класса list**

**//**

**iterator erase (iterator i ); // удаляет элемент, на который указывает данный  
// итератор. Возвращает итератор элемента, следующего за удаленным.**

**iterator erase (iterator st, iterator fin); // удалению подлежат все элементы  
// между st и fin, но fin не удаляется. Возвращает fin.**

**Iterator insert ( iterator i , const T& value = T()); // вставка некоторого  
// значения value перед i. Возвращает итератор вставленного элемента).**

**void insert (iterator i , size\_type n, const T&value); // вставка n копий  
// элементов со значением value перед i.**

**void push\_back ( const T&value ) ; // добавляет элемент в конец списка**

**void push\_front ( const T&value ) ; // добавляет элемент в начало списка**

**void pop\_back ( ) ; // удаляет последний элемент (не возвращает значение!)**

**void pop\_front ( ) ; // удаляет первый элемент списка**

**size\_type size( ) const; // выдает количество элементов списка**

**bool empty ( ) const; // возвращает истину, если вызывающий список пуст**

**void clear( ); // удаляет все элементы списка**

**.....**

**}**



# Пример использования STL

Функция, формирующая по заданному вектору целых чисел список из элементов вектора с четными значениями и распечатывающая его.

```
# include < iostream >
# include < vector >
# include < list >
using namespace std;

void g (vector <int> & v, list <int> & lst) {
    int i;
    for (i = 0; i < v.size(); i++)
        if ( ! ( v[ i ] % 2 ) )
            lst.push_back( v[ i ] );
    list < int > :: const_iterator p = lst.begin ();
    while ( p != lst.end ( ) ) {
        cout << *p << endl;
        p++;
    }
}
```

# Достоинства STL - подхода

- Каждый контейнер обеспечивает стандартный интерфейс в виде набора операций, так что один контейнер может использоваться вместо другого, причем это не влечет существенного изменения кода.
- Дополнительная общность использования обеспечивается через стандартные итераторы.
- Каждый контейнер связан с распределителем памяти - аллокатором, который можно переопределить с тем, чтобы реализовать собственный механизм распределения памяти.
- Для каждого контейнера можно определить дополнительные итераторы и интерфейсы, что позволит оптимальным образом настроить его для решения конкретной задачи.
- Контейнеры по определению однородны, т.е. должны содержать элементы одного типа, но возможно создание разнородных контейнеров как контейнеров указателей на общий базовый класс.
- Алгоритмы, входящие в состав STL, предназначены для работы с содержимым контейнеров. Все алгоритмы представляют собой шаблонные функции, следовательно, их можно использовать для работы с любым контейнером.