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 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.

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:

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

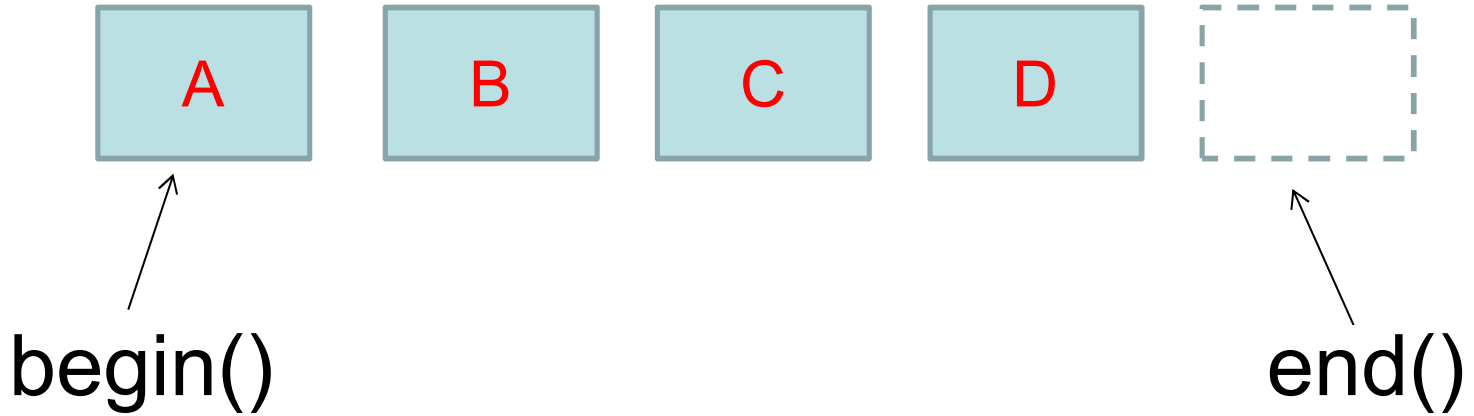
- Каждый контейнер обеспечивает свои итераторы, также поддерживающие стандартный набор итерационных операций со стандартными именами и смыслом. Итераторные классы и функции находятся в заголовочном файле <iterator>.
- **Итератор –** это класс, объекты, которого по отношению к контейнерам играют роль указателей, они, по сути, склеивают ядро STL в одну библиотеку.

Каждый контейнер содержит ряд ключевых функций-членов, позволяющих найти концы последовательности элементов в виде соответствующих значений итераторов:

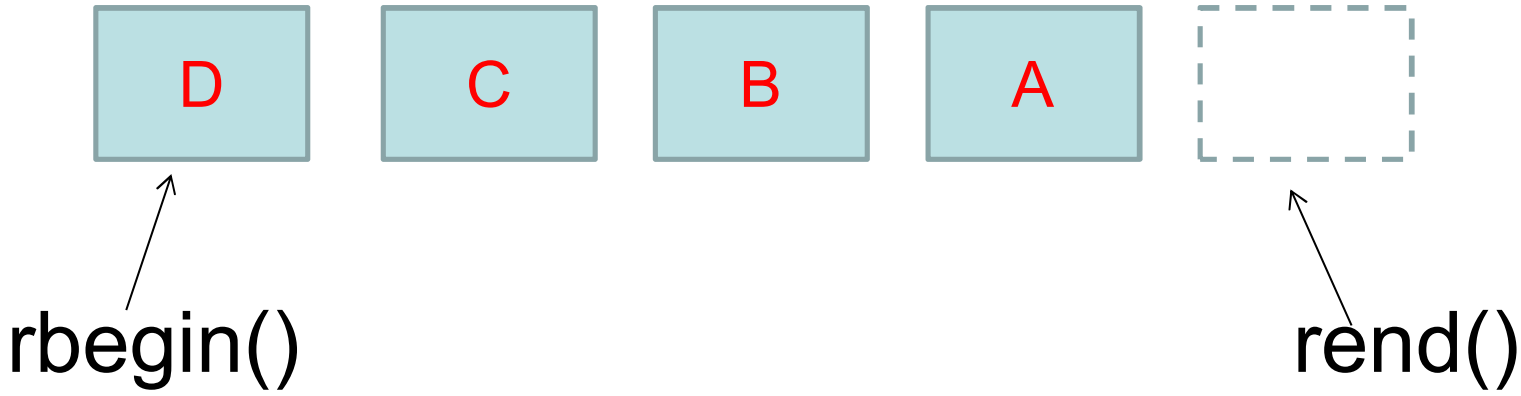
- **iterator begin(); –** возвращает итератор, который указывает на первый элемент последовательности.
- **const_iterator begin() const;**
- **iterator end(); –** возвращает итератор, который указывает на элемент, следующий за последним элементом последовательности (используется при оформлении циклов).
- **const_iterator end () const;**

- **reverse_iterator rbegin(); -** возвращает итератор, указывающий на первый элемент в обратной последовательности (используется для работы с элементами последовательности в обратном порядке).
- **const_reverse_iterator begin() const;**
- **reverse_iterator rend(); -** возвращает итератор, указывающий на элемент, следующий за последним в обратной последовательности.
- **const_reverse_iterator rend () const;**

- «прямые» итераторы



- «обратные» итераторы



- 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.

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

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

Пример: шаблонная функция **find()**. Нужен итератор, с какого начинается поиск, каким заканчивается и элемент, который мы ищем. Для целей функции достаточно итератора ввода (из контейнера).

```
Template < class InputInterator, 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.

Контейнер вектор

template < class T , class A = allocator < T > > class vector {

.....

public:

// **vector** – имя контейнера,

// **T** – тип элементов контейнера (value_type),

// **A** – распределитель памяти (allocator_type) - необязательный параметр.

// **Типы** - typedef….

// …….. - см. выше

// **Итераторы**

// …….. - см. выше

//

Контейнер вектор

// **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 вектор

// **Конструкторы и т.п.**

// конструкторы, которые могут вызываться с одним параметром , во избежание случайного

// преобразования объявлены explicit, что означает,

// что конструктор может вызываться только явно (vector<int> v=10 - ошибка,

// попытка неявного преобразования 10 в vector<int>)

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

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

Контейнер вектор

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

~vector();

//……

Контейнер вектор

// **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).

Контейнер вектор

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

…….

}

Пример 1. Печать элементов вектора в прямом порядке.

```
# include < vector >;
using namespace std;
int main () {
    vector < int > V( 100,5 );
    vector < int > :: const_iterator  p = V.begin ();
    while (p != V.end ()) {
            cout << *p << ' ';
            p++;
    }
    cout << endl ;
}
```

Или в обратном порядке :

```
…
vector < int > :: const_reverse_iterator  q = V.rbegin ( );
while ( q != V.rend ()) {
            cout << *q << ' ';  // печать в обратном порядке //
            q++;
}
cout << endl;…
```

Пример 2. Формирование вектора из чисел от 0 до 9

```
int main () {
vector < int > V ; //  вектор 0-ой длины //
int i ;
for ( i = 0 ; i < 10 ; i++ )  V.push_back (i) ;
cout << V.size () << endl ; //10
// vector <int > :: iterator p=V.begin ( ) ;
// p+=2;  // p++ ; p++;
}
```

Контейнер список

```
template < class T , class A = allocator < T > > class list
    {
.......
public:
// list – имя контейнера,
// T – тип элементов контейнера (value_type),
// A – распределитель памяти (allocator_type) -
    необязательный параметр.

// Типы - typedef….
//  …….. - см. выше
// Итераторы
//  …….. - см. выше
//
```

Контейнер список

// **Доступ к элементам**

//

reference front ( ); // первый элемент вектора

const_reference front ( ) const;

reference back (); // последний элемент вектора

const_reference back ( ) const;

Контейнер список

// **Конструкторы и т.п.**

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

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

Контейнер список

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; в этом случае конструктор

// умолчания в классе T обязателен)

template <class I> list (I first, I last, const A& = A()); // инициализация списка копированием элементов из [first, last), I - итератор для чтения

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

~ list();

//……

Контейнер список

// **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).

Контейнер список

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 pop_back () ; // удаляет последний элемент (не возвращает значение!)

void push_front (const T&value) ; // добавляет элемент в начало списка

void pop_front () ; // удаляет первый элемент

size_type size() const; // выдает количество элементов списка

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

void clear(); //удаляет все элементы списка

…….**}**

Пример 1: написать функцию, добавляющую в конец списка вещественных чисел элемент, значение которого равно среднему арифметическому всех его элементов.

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

void g (list <double> &lst) {
    list < double > :: const_iterator  p = lst.begin ();
    double s = 0; int n;
    while ( p!=lst.end ()) {
            s = s+*p;  n++;  p++;
    }
    if (n != 0) lst.push_back (s/n); // lst.push_back (s/lst.size());
}
```

Пример 2: написать функцию, формирующую по заданному вектору целых чисел список из элементов вектора с четными значениями и распечатывающую 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++;
    }
}
```

```
int main () {
   vector < int > v(20);  list < int >  lst;  int i;
   for (i=0; i<20; i++) v[i] = i;
cout << "vector is created" <<endl;
   g (v, lst);
   return 0;
}
```

Пример 3: написать функцию, формирующую по заданному списку целых чисел вектор из элементов списка с четными значениями и распечатывающую его.

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

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

```
int main () {
   vector < int > v(20);  list < int >  lst;
   for (i=0; i<20; i++) lst.push_back(i);
cout << "list is created" <<endl;
   g (v, lst);
   return 0;
}
```

Достоинства и недостатки 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
- **-** контейнеры не имеют фиксированного стандартного представления. Они не являются производными от некоторого базового класса. Это же верно и для итераторов. Использование стандартных контейнеров и итераторов не подразумевает никакой явной или неявной проверки типов во время выполнения
- **-** каждый доступ к итератору приводит к вызову виртуальной функции. Эти затраты по сравнению с вызовом обычной функции могут быть значительными
- **-** предотвращение выхода за пределы контейнера по-прежнему возлагается на программиста, при этом каких-то специальных средств для такого контроля не предлагается.