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:

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

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.

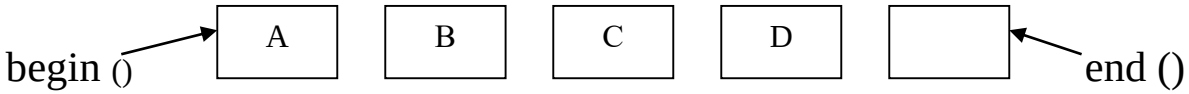
Итераторные классы и функции находятся в заголовочном файле

< iterator >.

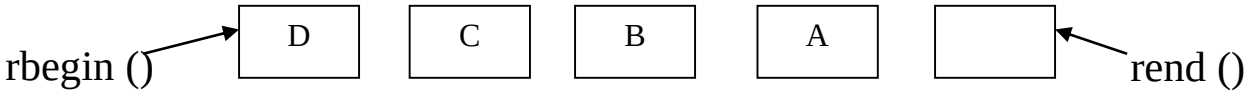
Методы контейнеров для нахождения значений итераторов концов последовательности элементов

- **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 какого 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 = v.begin ();              // Err!
    typename vector <T> :: iterator i = v.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*.

Iterat*or 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

bo*ol 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 f*in*, 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 пу**ст**
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.

Контейнеры специального вида. Пример 1.

```
#include <iostream>          // Считать строку слов до '\n',
#include <string>            // поместить отдельные слова строки
#include <sstream>           // в вектор, распечатать элементы
#include <vector>            // вектора
int main ()      {      std::string str, word;
                        getline(std::cin, str);
                        std::vector <std::string> vecstr;
                        std::istringstream ss (str);
                        while (ss >> word)
                                vecstr.push_back(word);
                        int  vsize = vecstr.size();
                        for (int i = 0; i < vsize; i++)
                        std::cout << vecstr[i] << std::endl;
                        return 0;  }
```

Контейнеры специального вида. Пример 2.

```
#include <iostream>   // Инициализировать строку символами-цифрами,
#include <string>     // поместить отдельные числа в переменные типа
#include <sstream>    // int, увеличить значения на 1, собрать обратно в
                      // строку и напечатать
int main ()    {
        std::string s = "123  456";
        std::istringstream   is (s);
        int a, b;
        is >> a >> b;
        std::ostringstream   os;
        os << a+1 << "  " << b+1;
        std::cout << os.str() << std::endl;    //  на экране:  124  457
        return 0;
}
```