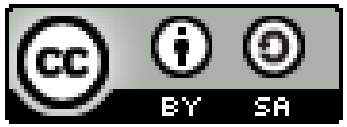
STL

Standard Template Library

Стандартная библиотека C++

Это произведение доступно по лицензии
Creative Commons “Attribution-ShareAlike” (“Атрибуция — На тех же условиях”) 3.0 Непортированная.
http://creativecommons.org/licenses/by-sa/3.0/deed.ru



June 6, 2018

Требования к стандартной библиотеке

Без наличия стандартной библиотеки пользоваться языком невозможно.

Требования к стандартной библиотеке:

- Универсальность.
- Ортогональность.
- Полнота.
- Удобство.
- Эффективность.
- Совместимость с языком C.

Состав STL

STL состоит из следующих компонент:

- Функции и типы из стандартной библиотеки языка C.
- Контейнеры и итераторы.
- Алгоритмы.
- Ввод-вывод.
- Строки.
- Средства поддержки языка.
- Средства обработки ошибок.
- ...

Функции и типы из стандартной библиотеки языка C.

Следствие требования совместимости с языком C.

Программа на языке C, если она не использует неправильно ключевые слова C++, должна компилироваться в C++. Следовательно, большинство функций из стандартной библиотеки C должны быть в STL.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
int main()
{
  int a; double b, c;
  printf( "Input␣a␣and␣b\n" );
  scanf( "%d␣%lf", &a, &b );
  c = exp( sin( b ));
  printf( "c␣=␣%f\n", c );
  return 0;
}
```

Контейнеры и итераторы

Контейнер — это объект, который содержит другие объекты (элементы) и предоставляет к ним доступ.

Примерами контейнеров служит вектор (**vector**), список (**list**), множество (**set**), ассоциативный массив (**map**), массив языка C.
Контейнеры бывают стандартные (**vector**, **map** . . .) и специальные (**string**, **rope** . . .).

Итератор — это объект, который ведёт себя подобно указателю на элементы контейнера.

Для итератора определён оператор разыменования (унарный *), и соответствующие контейнеру действия, позволяющие перемещаться по его элементам. У каждого контейнера — свои итераторы.

Свойства контейнеров

Каждый контейнер оптимизирован для определённого набора задач. Например, **vector** используется как современная замена массивов C, и предоставляет быстрый произвольный доступ к своим элементам. Контейнер **list** используется тогда, когда необходимо часто вставлять и удалять элементы в произвольную позицию.
Контейнеры бывают:

- **Последовательные**, т.е. хранящие и позволяющие перебрать все элементы в определённой последовательности. Примеры: **vector, array, list, deque, string**.
- **Ассоциативные**, т.е. предоставляющие доступ к своим элементом по заданному ключу. Внутренний порядок хранения стандартом не определён. Примеры: **map, set, multimap, unordered_map**.

Классификация итераторов (1)

Свойства итераторов определяются контейнером или потоком ввода/вывода. Итераторы бывают

Ввода Позволяет прочитать последовательность один раз в одном направлении. Пример: istream_iterator. Операторы: ++.

Вывода Позволяет записать последовательность один раз в одном направлении. Пример: ostream_iterator. Операторы: ++.

Однонаправленные Позволяет пройти (читать/писать) последовательность произвольное число раз в одном направлении. Операторы: ++.

Классификация итераторов (2)
продолжение

Двунаправленные Позволяет смещаться в контейнере на один элемент в любом направлении. Пример: **list**<T>::**iterator**. Операторы: ++ , −−.

Произвольного доступа Позволяет смещаться в контейнере на произвольное количество элементов в любом направлении. Пример: **vector**<T>::**iterator**. Операторы: ++ , −−, +=(**int**), −=(**int**) +(**int**) −(**int**) < > <= >= .

Контейнер vector

Предназначен для использования вместо массивов языка C. Итераторы — произвольного доступа. Определён оператор [] для произвольного доступа к элементам. Определены функции `push_back`, `pop_back` для добавления и удаления элементов в конец контейнера. Преимущества перед массивами C:

- Автоматическое управление памятью. Конструкторы создают массивы требуемого размера (пустые, копии других векторов, часть последовательности). При выходе за область видимости деструктор автоматически удаляет элементы и освобождает память. При добавлении – может заново выделить блок памяти.
- Есть функции `size`, `capacity`, позволяющие узнать размер и ёмкость. Распределить память заранее можно с помощью функции `reserve`.
- Не уступают в скорости массивам C.

Контейнер vector

Определения синонимов типов

Набор определений (упрощённое представление):

```
template< typename T, typename A = allocator<T> >
 class vector {
  typedef T value_type;
  typedef A allocator_type;
  typedef typename A::size_type size_type;
  // ...
  typedef ..... iterator;        // может быть T*
  typedef ..... const_iterator; // .... const T*
};
```

Используются для упрощения дальнейших определений и как синонимы для вспомогательных типов:

```
  vector<int> a;
  a.push_back(50); a.push_back(10);  a.push_back(5);
  for( vector<int>::iterator i = a.begin();
       i != a.end(); ++i )
    cout << *i << ' ';
```

Контейнер vector

C++11: использование auto и нового for

Предыдущий пример в C++11 можно упростить:

```
vector<int> a;
a.push_back(50); a.push_back(10);  a.push_back(5);
for( auto i = a.begin(); i != a.end(); ++i )
  cout << *i << ' ';
```

И даже так:

```
vector<int> a { 50, 10, 5 };
for( auto v : a )
   cout << v << ' ';
for( auto &v : a )
   v += 3;
```

Контейнер vector

Создание, уничтожение, копирование

Набор конструкторов, деструктор ...(упрощённое представление):

```
template< typename T > class vector {
  vector(); // по умолчанию
  explicit vector( size_type n, const T& val = T() );
  vector( const vector& other );
  vector( vector &&other );
  template<typename In>
    vector<T>::vector( In first, In last );
  vector( std::initializer_list<T> init );
  ~vector();

  vector& operator=( const vector &rhs );
  vector& operator=( vector &&rhs );

  void assign( size_type count, const T& value );
  void assign( std::initializer_list<T> ilist );
  template< class InputIt >
    void assign( InputIt first, InputIt last );
};
```

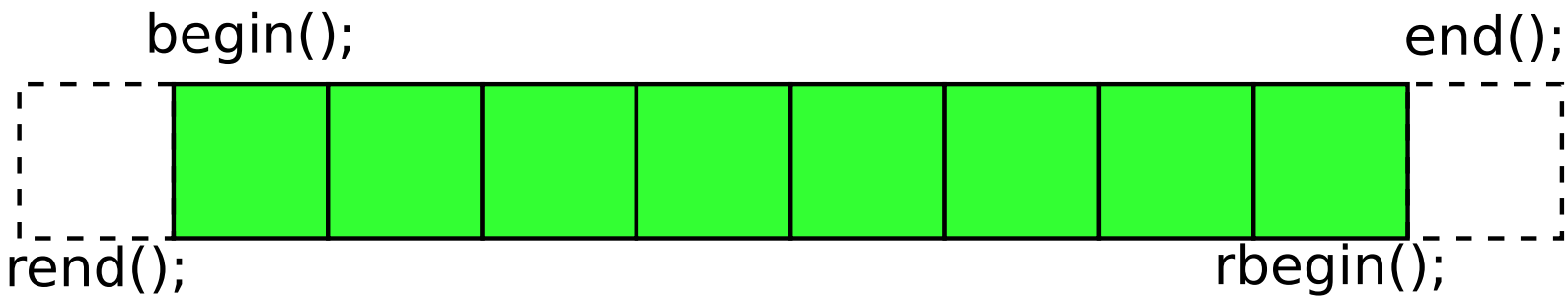
Контейнер vector

Примеры использования конструкторов, присваивания

```
vector<double> mx;
vector<int> ma(100);
vector<double> mz { 5.1, 1e7, 14, -5e-4 };
vector<int> mb = ma;
ma = mb;

vector<XX> xx( 5, XX(1,2,"z") );
vector<int> mc( ma.begin(), ma.end() );

vector<string> as { "first", "word", "and", "second" };

mz.assign( 10000, M_PI );
```

Функции-члены для получения итераторов.

```
template< typename T > class vector {
  iterator begin(); // начало контейнера
  const_iterator cbegin() const;
  iterator end(); // ЗА концом контейнера
  const_iterator cend() const;
  iterator rbegin(); // начало в обратнном напр.
  const_iterator crbegin() const;
  iterator rend(); // перед началом контейнера
  const_iterator crend() const; // .....
};
```



Оператор [ ] и функция at()

```
template< typename T > class vector {
  T& operator[](size_t n);
  const T& operator[](size_t n) const;
  T& at(size_t n);  // + std::out_of_range
  const T& at(size_t n) const;
  T& front(); // первый элемент
  T& back(); // последний элемент
};
```

Примеры использования:

```
vector<int> ma(100);
ma[5] = ma[0] + 12; ma.at(105) = 4;
ma.first() = 42;
```

Контейнер vector

Операции с концом контейнера (стек)

В вектор “дешево” добавлять и удалять элементы только в конце:

```
template< typename T > class vector {
  void push_back( const T& obj );
  void pop_back();
};
```

Примеры использования:

```
vector<int> ma;      // [ ]
ma.push_back(100);   // [100]
ma.push_back(500);   // [100 500]
ma.push_back(900);   // [100 500 900]
ma.pop_back();       // [100 500]
ma.pop_back();       // [100]
```

Контейнер vector

Операции с произвольным местом

Можно, но дороже добавлять и удалять в произвольном месте:

```
template< typename T > class vector {
  iterator insert( iterator pos, const T& x );
  iterator insert( iterator pos, size_type n,
                   const T& x );
  template<class In>
    void insert(iterator pos, In first, In last );
  iterator erase(iterator pos);
  iterator erase(iterator first, iterator last);
  void clear();
};
```

Примеры использования:

```
vector<int> ma; ma.push_back(1); ma.push_back(5);
vector<int>::iterator i =  // [1 5]
  find( ma.first(), ma.last(), 1 );
if( i != ma.end() )
  ma.insert( i, 7 ); // [7 1 5]
```

Контейнер vector

Размер и ёмкость

Количество выделенной памяти обычно больше размера:

```
template< typename T > class vector {
  size_type size() const;       // колво– элементов
  bool empty() const;
  void reserve(size_type n);  //зарезервировать место
  size_type capacity() const; // место в( элементах)
  void resize( size_type sz, T val = T() );
  void shrink_to_fit();         // C++11
  void clear();
};
```

Примеры использования:

```
vector<int> ma;
ma.push_back(1); ma.push_back(5); ma.push_back(0);
cout << ma.size() < ' ' << ma.capacity() << endl;
ms.reserve(100);
cout << ma.size() < ' ' << ma.capacity() << endl;
```

Есть ещё == , < , swap ...

Пример использования vector

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

int main()
{
  vector<int> a;
  a.push_back(50); a.push_back(10);  a.push_back(5);
  a[1] = -7;
  cout << a.size() << ' ' << a.capacity() << endl;
  //        3                    4
  a.pop_back();
  for( vector<int>::iterator i = a.begin();
       i != a.end(); ++i )
    cout << *i << ' ';
  return 0;
}
```

Пример использования vector: C++11

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
  vector<int> a { 50, 10, 5, 17 };
  a.push_back( 42 );
  a[1] = -7;
  cout << a.size() << ' ' << a.capacity() << endl; // 5 8
  a.pop_back();
  for( auto &v : a ) {
    v *= 2;
  }
  for( auto v : a ) {
    cout << v << ' ';
  }
  int s = 0, n = 0;
  for_each( a.begin(), a.end(), [&s](int v){s+=v;} );
  cout "s= " << s << endl;
  generate( a.begin(), a.end(), [&n]{ return n++; });
  return 0;
}
```

Контейнер list — двусвязный список

Предназначен для работы в тех случаях, когда часто происходит вставка и удаление элементов в середине последовательности. Итераторы — двунаправленные.

Произвольного доступа и оператора `[]` нет.

Есть стандартный набор конструкторов и деструктор. Есть работающие за константное время функции добавления элемента[тов] перед указанным (insert(p,x); insert(p,first,last)) и удаления элементов (erase(p); erase(first,last)).

Функции splice, merge позволяют быстро вставлять один список в другой и объединять их.

Пример использования контейнера list

```
#include <list>
#include <iostream>
using namespace std;
int main()
{
  list<int> a;
  a.push_back(50); a.push_front(10); a.push_back(5);
  list<int>::iterator i = a.begin(); // auto i = ...
  ++i;  *i = -7;
  a.insert( i, 500 );
  i = a.end(); --i;
  a.insert( i, -100 );
  cout << a.size() << endl; // 5
  a.pop_back(); // size = 4
  for( auto i : a ) //
    cout << *i << ' ';
  cout << endl;
  return 0;
}
```

Другие последовательные контейнеры

Также используются:

- **deque** – дек — аналогичен vector, плюс возможность быстрых операций с началом (push_front, pop_front).
- **stack** — стек (LIFO), обычно сделан из list.
- **queue** — очередь (FIFO).
- **priority_queue** — очередь с приоритетами.
- **array** — массивы фиксированного размера (C++11).
- **forward_list** — односвязный список (C++11).

Ассоциативные контейнеры

Позволяет создавать контейнеры, для доступа к элементам которых может использоваться ключ — объект произвольного типа, для которого существует оператор <.

```
#include <map>
#include <iostream>
using namespace std;
int main()
{
  map<string,int> m;
  m["Vasya"] = 50;
  m["Katya" ] = 100;
  cout << m["Vasya"] << ' ' << m["Katya"];
  map<string,int>::iterator i;
  for( i = m.begin(); i!=m.end(); ++i; )
    cout << i->first << ' ' i->second << endl;
  return 0;
}
```

Специальный контейнер для строк — basic_string

Для работы со строками в C++ следует использовать string — специализацию шаблона basic_string для обычных символов. Помимо функций, обычных для контейнеров, в нём реализовано большинство функций, необходимых для строк.

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
  string a, b = "Vasya", c = b;
  a = "Katya";
  a += '!';   c = a + b;   c += "␣....";
  cout << a << '␣' << b << '␣' << c << endl;
  cout << c.length();
  const char *s = a.c_str();
  return 0;
}
```

More: clear, insert, compare, replace, substr, find..., to_string, stoi

Пример без использования string

```
class Stud {
 public:
   Stud( const char *aname, int anz );
   Stud( const Stud &r );
   Stud( Stud &&r );
   ~Stud() {delete[] name}; // ............
 private:
   int nz; char *name;
}
Stud::Stud( const char *aname, int anz )
     : nz(anz), name(new char[strlen(aname)+1] )
{
  strcpy( name, aname );
}
Stud::Stud( const Stud &r )
     : nz(r.nz), name(new char[strlen(r.name)+1] )
{
  strcpy( name, r.name );
}
```

Пример с использованием string

```
class Stud {
 public:
   Stud( const string &aname, int anz ); //...
 private:
   int nz; string name;
}
Stud::Stud( const string &aname, int anz )
     : nz(anz), name(aname)
{
}
```

Список контейнеров

array, vector, deque, list, forward_list, valarray

set, multiset, map, multimap, unordered_set, unordered_multiset, unordered_map, unordered_multimap

stack, queue, priority_queue

Вспомогательные и специальные:

pair, tuple, basic_string, bitset

Алгоритмы

Для работы с данными используется множество абстрактных алгоритмов. Заголовок — <algorithm>
Алгоритмы делятся на

- Немодифицирущие (поиск, подсчет, сравнение);
- Модифицирующие (копирование, изменение, обмен).
- Специальные (для определённых контейнеров).
- Всякие (min, max, min_element)

Всего — порядка 90 алгоритмов. Большинство в качестве аргументов требуют итераторы и функции/функторы.

for_each — выполнить действия для каждого элемента в последовательности;

find — найти элемент по значению;

find_if — найти элемент по условию;

find_first_of — найти элемент из одной последовательности в другой;

count — подсчёт по значению;

count_if — подсчёт по условию;

search_n — поиск *n*-ного вхождения элемента;

...

sort — сортировка;
copy — копирование
transform — изменение;
assign — присвоение;
swap — обмен;
replace — замена;
replace_if — замена по условию;
generate — заменить результатом операции;
reverse — поменять порядок на противоположный;
...

Алгоритм find

Условный синтаксис:

iterator_in find(iterator_in start, iterator_in end, T value);

Пример:

```
vector<int> a;
for( int i=0; i<10; ++i )
  a.push_back( 3 << i );
vector<int>::iterator j =find(a.begin(),a.end(),96);
if( j != a.end() )
  cout << *j << endl;

const char s[] = "ABCDEFGHIJKLMN";
const char *p = find( s, s+strlen(s), 'I' );
```

Алгоритм find_if

Условный синтаксис:

iterator_in find(iterator_in start, iterator_in end, condition);

Пример:

```
inline bool g50(int x) { return x > 50; }

vector<int> a;
for( int i=0; i<10; ++i )
  a.push_back( 3 << i );

vector<int>::iterator j
  =find_if( a.begin(), a.end(), g50 );
auto j1 = find_if( a.begin(), a.end(), // C++11
                   [](double v){return v > 50} );

if( j != a.end() )
  cout << *j << endl;
```

Ввод/вывод

Для ввода/вывода используется целая иерархия шаблонов и их специализаций. Объекты типа **ostream** предназначены для вывода, **istream** — для ввода, **iostream** — для ввода и вывода. Это — абстрактные классы (специализации шаблонов), так как для них не задано, с чем будет происходить обмен данными.
Для работы с файлами используются **ofstream, ifstream, fstream**. При выходе объекта за пределы области видимости закрытие файлов происходит автоматически.
Для работы со строками используются **ostringstream, istringstream, stringstream**.
Для всех фундаментальных типов определены операторы >> — для ввода и << — для вывода. Для пользовательских типов — можно определить самим.
Управлять потоком позволяют манипуляторы.

Пример использования потоков ввода/вывода

```
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;
int main()
{
  int a; double b; char c; string s;
  ifstream is( "in_file.txt" );
  is >> a >> b >> c >> s;
  ofstream os( "out_file.txt" );
  os << setw(10) << a << setprecision(4) << b
     << c << '␣' << s << endl;
  return 0;
}
```

Средства обработки ошибок

Для сигнализации об ошибках в STL используются исключения. Базовым классом для всех ошибок STL является **exception**. От него наследуются

length_error — недопустимая длина;

domain_error — недопустимое входное значение;

invalid_argument — неправильный аргумент;

bad_alloc — ошибка выделения памяти;

out_of_range — недопустимый индекс . . .

More

Дата и время: **duration, system_clock, chrono::time_point**.

Функции и привязки: **function, bind, ref**.

Численные алгоритмы:
iota, accumulate, inner_product, adjacent_difference, partial_sum, **template< class T > class complex, linear_congruential_engine**.

Локализация: **locale, use_facet, isspace**, . . .

Регулярные выражения: **basic_regex, sub_match, match_results** . . .

Работа с нитями выполнения: **thread, yield, mutex, promise, future** . . .

Атомарные операции: **atomic, atomic_store, atomic_flag** . . .