STL: ассоциативные контейнеры и итераторы

Александр Смаль

CS центр
22 февраля 2017
Санкт-Петербург

STL: введение

- STL = Standard Template Library
- STL описан в стандарте C++, но не упоминается там явно.
- Авторы: Александр Степанов, Дэвид Муссер и Менг Ли для HP, а потом для SGI.
- Основан на разработках для языка Ада.
- Основные составляющие:
  - контейнеры (хранение объектов в памяти),
  - итераторы (доступ к элементам контейнера),
  - алгоритм (для работы с последовательностями),
  - адаптеры (обёртки над контейнерами)
  - функциональные объекты, функторы (обобщение функций).
  - потоки ввода/вывода.
- Всё определено в пространстве имён `std`.

Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на четыре категории:

- последовательные,
- **ассоциативные**,
- контейнеры-адаптеры,
- псевдоконтейнеры.

Требования к хранимым объектам:

1. copy-constuctable
2. assignable
3. “стандартная семантика”

Итераторы — объекты для доступа к элементам контейнера с синтаксисом указателей.

Общие члены контейнеров

Типы (typedef-ы или вложенные класс):

1. `C::value_type`
2. `C::reference`
3. `C::const_reference`
4. `C::pointer`
5. `C::iterator`
6. `C::const_iterator`
7. `C::size_type`

Методы:

1. Конструктор по умолчанию, конструктор копирования, оператор присваивания, деструктор.
2. `begin()`, `end()`
3. Операторы сравнения: `==`, `!=`, `>`, `>=`, `<`, `<=`.
4. `size()`, `empty()`.
5. `swap(obj2)`

Ассоциативные контейнеры

Общие методы:

1. `erase` по key
2. `count`
3. `find`
4. `lower_bound, upper_bound, equal_range`
5. `insert` с подсказкой

Особенности:

1. Требуют отношение порядка.
2. Нет произвольного доступа.

set, multiset

```
std::set<int> primes;
primes.insert(2);
primes.insert(3);
primes.insert(5);
...
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

for(std::set<int>::iterator it  = primes.begin();
                            it != primes.end(); ++it)
    std::cout << *it << '\n';
// ----------------
std::multiset<int> ms;
ms.insert(1);
ms.insert(2);
ms.insert(2); // ms.size() == 3
std::cout << ms.count(2) << '\n';
```

map, multimap

Хранит пару ключ-значение `std::pair`.

```
template<class F, class S>
struct pair {
    ... // constructors
    F  first;
    S  second;
};
template<class F, class S>
pair<F, S> make_pair(F const& f, S const& s);

template<class Key, class T, ...> class map {
...
typedef pair<const Key, T> value_type;
} ;
```

Особые методы:

- `operator[]`

map, multimap

```
std::map<string,int> phonebook;
phonebook.insert(std::make_pair("Mary", 2128506));
phonebook.insert(std::make_pair("Alex", 9286385));
phonebook.insert(std::make_pair("Bob",  2128506));
...
std::map<string,int>::iterator it = phonebook.find("John");
if ( it != phonebook.end())
    std::cout << "Jonh's p/n is " << it->second << "\n";

for(it = phonebook.begin(); it != phonebook.end(); ++it)
    std::cout << it->first << ": " << it->second << "\n";
// ----------------
std::multmap<string, int> pb;
pb.insert(std::make_pair("Mary", 2128506));
pb.insert(std::make_pair("Mary", 2128507));
pb.insert(std::make_pair("Mary", 1112223)); //ms.size()==3
std::cout << pb.count("Mary") << '\n';
```

map::operator[]

```
std::map<string, int> phonebook;
phonebook.insert(std::make_pair("Mary", 2128506));
...
phonebook.insert(std::make_pair("Mary", 2128507)); // fail

std::pair<std::map<string, int>::iterator, bool> res =
    phonebook.insert("Mary", 2128507); // res.second == false

std::map<string, int>::iterator it = phonebook.find("Mary");
if (it != phonebook.end() )
     it->second = 2128507;
else phonebook.insert(std::make_pair("Mary", 2128507));
// OR
phonebook["Mary"] = 2128507;

for(it = phonebook.begin(); it != phonebook.end(); ++it)
    std::cout << it->first << ": " << phonebook[it->first] << "\n";
```

Ограничения `map::operator[]`

1. Работает только с неконстантным map.
2. Требует наличие конструктора по умолчанию у `T`.

```
T & operator[](Key const& k)
{
    iterator i = find(k);
    if (i == end())
        i = insert(value_type(k, T())).first;

    return i->second;
}
```

3. Работает за $O(\log n)$. $\Rightarrow$ Не стоит работать с `map` как с массивом!

Удаление из set и map

Неправильный вариант

```
std::map<string, int> m;
std::map<string, int>::iterator it = m.begin();
for( ; it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

Правильный вариант

```
for( ; it != m.end(); )
    if (it->second == 0) m.erase(it++);
    else ++it;
```

C++ 11

```
for( ; it != m.end(); )
    if (it->second == 0) it = m.erase(it);
    else ++it;
```

Использование собственного компаратора

```
struct Person {
    string name;
    string surname;
};
bool operator<(Person const& a, Person const& b) {
    return a.name < b.name ||
           (a.name == b.name && a.surname < b.surname);
}
std::set<Person> s1; // unique by name+surname

struct PersonComp {
    bool operator()(Person const& a, Person const& b) const {
        return a.surname < b.surname;
    }
};

std::set<Person, PersonComp> s2; // unique by surnames
```

Требования к компаратору

Компаратор должен задавать отношение строгого порядка:

$$\neg(x \prec y) \wedge \neg(y \prec x) \Rightarrow x = y$$

insert с подсказкой

```
std::map<K, V> m;

K k = ...;
V v = ...;

std::map<K, V>::iterator i = m.find(k); // returns m.end()

std::map<K, V>::iterator hint = m.lower_bound(k);
if (hint != m.end() && !(k < hint->first))
    // gotcha!
else
    // use hint
    m.insert(hint, std::make_pair(k, v));
```

Категории итераторов

Итератор — синтаксически похожий на указатель объект для доступа к элементам последовательности.
Итераторы делятся на пять категорий.

- Random access iterator. ++, --, арифметика, read-write
- Bidirectional iterator. ++, --, read-write
- Forward iterator. ++, read-write
- Input iterator. ++, read
- Output iterator. ++, write

Функции для работы с итераторами:

```
void    advance  (Iterator & it, size_t n);
size_t distance (Iterator f, Iterator l);
void    iter_swap(Iterator i, Iterator j);
```

iterator_traits

```
// <iterator>
template <class Iterator>
struct iterator_traits {
    typedef Iterator::difference_type   difference_type;
    typedef Iterator::value_type        value_type;
    typedef Iterator::pointer           pointer;
    typedef Iterator::reference         reference;
    typedef Iterator::iterator_category iterator_category;
};

template<class Iterator>
void iter_swap(Iterator i, Iterator j) {
    // Iterator::value_type t = *i;
    typename iterator_traits<Iterator>::value_type t = *i;
    *i = *j;
    *j = t;
}
```

iterator_traits для указателей

```
// <iterator>

template <class T>
struct iterator_traits<T *> {
    typedef ptrdiff_t                   difference_type;
    typedef T                           value_type;
    typedef T*                          pointer;
    typedef T&                          reference;
    typedef random_access_iterator_tag  iterator_category;
};
```

iterator_category

```
struct bidirectional_iterator_tag {};
struct forward_iterator_tag {};
struct input_iterator_tag {};
struct output_iterator_tag {};
struct random_access_iterator_tag {};

template<class Iterator>
void advance(Iterator & i, size_t n) {
    advance_impl(i, n, typename iterator_traits<Iterator>::
                        iterator_category());
}
template<class Iterator>
void advance_impl(Iterator & i, size_t n, random_access_iterator_tag)
{ i += n; }

template<class Iterator>
void advance_impl(Iterator & i, size_t n, ... ) {
    for (size_t k = 0; k != n; ++k, ++i );
}
```

reverse_iterator

У стандартных контейнеров есть обратные итераторы:

```
list<int> l;
for(list<int>::reverse_iterator i  = l.rbegin();
                                i != l.rend(); ++i)
    ...
```

Конвертация итераторов:

```
iterator           i;
reverse_iterator ri = i;
i = ri.base();
```

Есть возможность сделать reverse итератор по RA или BiDi.

```
#include <iterator>

template <class Iterator>
class reverse_iterator;
```

Инвалидация итераторов

Некоторые операторы над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление инвалидирует все итераторы.
3. `deque` удаление/добавление инвалидирует все итераторы, кроме случаев удаления/добавления первого или последнего элементов.
4. В `vector`, если `capacity` $>$ `size`, то гарантируется, что при добавлении элемента инвалидируются только итераторы на все следующие за добавленным элементы.

Как написать свой итератор

```
#include <iterator>

template
<class Category,                    // iterator::iterator_category
 class T,                           // iterator::value_type
 class Distance = ptrdiff_t,        // iterator::difference_type
 class Pointer = T*,                // iterator::pointer
 class Reference = T&               // iterator::reference
> class iterator;

struct MyIterator
    : std::iterator<bidirectional_iterator_tag, Person>
{
// ++, --, ->, * ...
};
```