Standard Template Library

ЛЕКЦИЯ №9

STL

1. Входит в поставку стандартных C++ компиляторов
2. Содержит контейнеры, структуры данных, итераторы и алгоритмы
3. Спецификация находится тут: http://www.cplusplus.com/reference

Контейнеры

Контейнер — это объект, который может содержать в себе другие объекты. Существует несколько разных типов контейнеров. Например, класс vector определяет динамический массив, deque создает двунаправленную очередь, а list представляет связный список. Эти контейнеры называются **последовательными контейнерами** (sequence containers), потому что в терминологии STL последовательность — это линейный список.

STL также определяет **ассоциативные контейнеры** (associative containers), которые обеспечивают эффективное извлечение значений на основе ключей. Таким образом, ассоциативные контейнеры хранят пары “ключ/значение”. Примером может служить map. Этот контейнер хранит пары “ключ/значение”, в которых каждый ключ является уникальным. Это облегчает извлечение значения по заданному ключу.

Виды контейнеров

Контейнер	Описание	Требуемый заголовок
deque	Двунаправленная очередь	<deque>
list	Линейный список	<list>
map	Хранит пары «ключ/значение», где каждый ключ ассоциирован только с одним значением	<map>
bitset	Битовое поле — это массив битов фиксированного размера	<bitset>
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>
priority_queue	Очередь с приоритетами	<deque>
queue	Очередь	<deque>
set	Множество уникальных элементов	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

Операции, поддерживаемые контейнерами

Все контейнеры должны поддерживать операцию присваивания. Они должны также поддерживать все логические операции. Другими словами, все контейнеры должны поддерживать следующие операции:

=,==,<,<=,!=,>,>=

Все контейнеры должны иметь конструктор, создающий пустой контейнер, и конструктор копирования. Они должны предусматривать деструктор, который освобождает всю память, использованную контейнером, и вызывать деструктор каждого элемента в контейнере.

Итераторы

Одной из основных парадигм данной библиотеки было разделение двух сущностей: контейнеров и алгоритмов. Но для непосредственного воздействия алгоритмом на данные контейнера пришлось ввести промежуточную сущность — итератор.

Итераторы позволили алгоритмам получать доступ к данным, содержащимся в контейнере, независимо от типа контейнера. Но для этого в каждом контейнере потребовалось определить класс итератора. Таким образом алгоритмы воздействуют на данные через итераторы, которые знают о внутреннем представлении контейнера.

Что нового в C++: range-based циклы

Example64_RangeFor

В C++11 была добавлена поддержка парадигмы foreach для итерации по набору. В новой форме возможно выполнять итерации в случае, если для объекта итерации перегружены методы begin() и end().

Данный циклы не работают с массивами, аллоцируемыми динамически (т.к. про них компилятор не знает, какой у них будет размер). А вот с такими работает:

```
int arr[] = { 1, 2, 3, 4, 5 };
for (auto e : arr)
std::cout << e << std::endl;
```

Простейший итератор для RangeFor
Example65_Iterator

```
// for работает с итератором как с указателем!
1.class IntIterator{
2.int operator*() ;
3.int operator->();
4.bool operator!=(IntIterator const& other)
 const;
5.IntIterator & operator++() ;
6.}
```

std::initializer_list<T> списки инициализации

```
#include <initializer_list>

struct myclass {
  myclass (int,int);
  myclass (initializer_list<int>);
  /* definitions ... */
};

myclass foo {10,20};  // calls initializer_list ctor
myclass bar (10,20);  // calls first constructor
```

Методы контейнеров

Все контейнеры должны предоставлять перечисленные ниже функции.

iterator begin()	Возвращает итератор, указывающий на первый элемент контейнера.
const_iterator begin() const	Возвращает константный итератор, указывающий на первый элемент контейнера.
iterator end()	Возвращает итератор, указывающий на позицию, сле- дующую за последним элементом контейнера.
const_iterator end() const	Возвращает константный итератор, указывающий на по- зицию, следующую за последним элементом контейнера.
bool empty() const	Возвращает true, если контейнер пуст.
size_type size() const	Возвращает количество элементов, в текущий момент хранящихся в контейнере.
void swap(*ContainerType* c)	Обменивает между собой содержимое двух контейнеров.

Требования к последовательному контейнеру

void clear()	Удаляет все элементы из контейнера.
iterator erase(iterator *i*)	Удаляет элемент, на который указывает *i*. Возвращает итератор, указывающий на элемент, находящийся после удаленного.
iterator erase(iterator *start*, iterator *end*)	Удаляет элементы в диапазоне, указанном *start* и *end*. Возвращает итератор, указывающий на элемент, нахо- дящийся после последнего удаленного.
iterator insert(iterator *i*, const T &*val*)	Вставляет *val* непосредственно перед элементом, спе- цифицированным *i*. Возвращает итератор, указываю- щий на вставленный элемент.
void insert(iterator *i*, size_type *num*, const T &*val*)	Вставляет *num* копий *val* непосредственно перед эле- ментом, специфицированным *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*)	Вставляет последовательность, определенную *start* и *end*, непосредственно перед элементом, специфициро- ванным *i*.

std::vector
#include <vector>

1. Эквивалент массива с динамическим размером
2. Параметры шаблона:
   `template < class T, class Alloc = allocator<T> > class vector;`
3. Итераторы:
   1. begin, end – обычные (cbegin, cend – константные)
   2. rbegin, rend – revers итераторы (crbegin, crend – константные)
4. Доступ к элементам: [size_t] , at(size_t)
5. Модификаторы:
   1. push_back, pop_back –добавление/удаление элемента в конец
   2. insert – добавление элемента в произвольную точку

Пример
Example70_vector

```
1.// vector размера 5 заполненыый int со значением 100
2. std::vector<int> v(5, 100);

3.// печать всех элементов
4. for (int i : v) std::cout << i << std::endl;

5.// итератор для вектора
6.std::vector<int>::iterator it;

7.// удаляем элементы равные 100
8. for (it = v.begin(); it != v.end();) {
9.         if (*it == 100) // значение элемента на итераторе
10.                  v.erase(it); // удаление элемента
11.         else it++; // сдвиг итератора
12.    };
```

std::map
#include <map>

1. Используются для создания «словарей», т.е. пар ключ-значение.

```
template < class Key, // map::key_type
      class T, // map::mapped_type
      class Compare = less<Key>, // map::key_compare
      class Alloc = allocator<pair<const Key,T> > // map::allocator_type
      > class map;
```

2. Основное назначение – поиск значений по ключу. Делается это с помощью
   `std::pair< map::key_type, map::mapped_type> operator[](map::key_type)`
3. В pair есть два атрибута– first и second
4. pair описан в #include<utility>

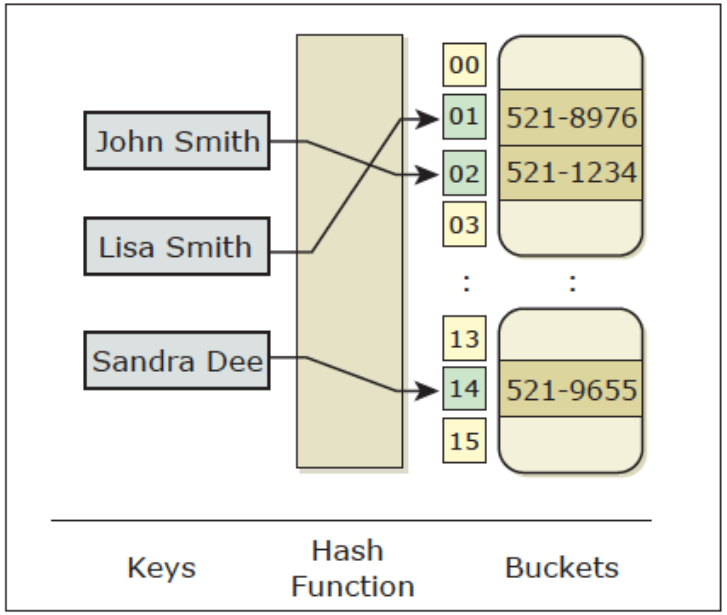

Image by MIT OpenCourseWare.

Пример
Example71_map

```
1.// пара строка – целое число
2.std::map<const char*,int> age;

3.// добавляем или меняем значение
4.age["Иванов"] = 18;

5.// явное добавление
6.age.insert(std::pair<const char*,int>("Петров",21);

7.// удаляем значение
8.age.erase("Петров");

9.// печатаем на экран с помощью итератора
10.for(auto value:age)
11.std::cout << "Age of " <<value.first << "=" << value.second << std::endl;
```

std::deque
Example79_Deque

```
1.std::deque<std::string> myDeque; // создаем пустой дек типа string
2.myDeque.push_back(std::string("World ")); // добавили в дек один элемент типа
  string
3.std::cout << "Количество элементов в деке: " << myDeque.size() << std::endl;
4.myDeque.push_front("Hello "); // добавили в начало дека еще один элемент
5.myDeque.push_back(" !!!"); // добавили в конец дека элемент
6.std::cout << "Количество элементов в деке изменилось, теперь оно = " <<
7. myDeque.size() << std::endl;
8.std::cout << "\nВведенный дек: ";
9. for(auto i:myDeque) std::cout << i << " ";
10.myDeque.pop_front(); // удалили первый элемент
11.myDeque.pop_back(); // удалили второй элемент
```

std::set
Example80_Set

```
1.#include <iostream>
2.#include <set>  // заголовочный файл множеств и мультимножеств
3.#include <iterator>
4.int main(){
5.std::set<char> mySet; // объявили пустое множество

6.// добавляем элементы в множество
7.mySet.insert('I’);
8.// печатаем
9.for( auto i: mySet)  std::cout << i << " ";
10.std::cout << std::endl;
11.// удаляем i
12.mySet.erase('i');
13.// ищем I
14. if(mySet.find('I')!=mySet.end()) std::cout << "Found I" << std::endl;
15.return 0;
16.}
```

std::bitset
Example81_bitset

```
1.#include <iostream>
2.#include <bitset>   // заголовочный файл битовых полей
3.int main( ) {
4.     int number;
5.     std::cout << "Введите целое число от 1 до 255: ";
6.     std::cin >> number;
7.     std::bitset<8> message(number);
8.     std::cout << number << " = " << message << std::endl;

9.      std::bitset<8> bit2 = message;
10.     message = message.flip();
11. // поменять все биты на противоположные
12.     std::cout << "Инвертированное число: " << message << std::endl;
13.     std::bitset<8> bit3 = bit2 | message;
14.}
```

Итераторы

ОДНОТИПНАЯ РАБОТА С РАЗНОТИПНЫМИ КОНТЕЙНЕРАМИ

Итераторы в <vector>

1. Это объект, который указывает на элемент в структуре данных и позволяет перебирать элементы этой структуры данных. Фактически он действует как указатель на элемент массива.
2. Для вектора он выглядел так:
   `std::vector<int>::iterator it;`
3. Доступ к элементам мы осуществляли:`*it;`
4. Смещение по итератору производили `it++, it--;`

Итератор

Контейнер может иметь произвольную структуру и различные методы доступа:

Итератор указывает на элемент контейнера и знает как перейти к следующему элементу

Программе работает только с итератором и его интерфейсом (++)

Предопределенные итераторы

http://www.cplusplus.com/reference/iterator/

reverse_iterator

move_iterator

back_insert_iterator

front_insert_iterator

insert_iterator

istream_iterator

ostream_iterator

istreambuf_iterator

ostreambuf_iterator

std::back_insert_iterator
Example72_back_insert_iterator

```
1.std::vector<int> foo, bar;
2.    for (int i = 1; i <= 5; i++) {
3.        foo.push_back(i);
4.        bar.push_back(i * 10);
5.    }

6.// итератор добавляющй элементы в конец foo
7.std::back_insert_iterator< std::vector<int> > back_it(foo);

8.// копируем из bar в back_it
9.std::copy(bar.begin(), bar.end(), back_it);
```

Как устроен back_insert_iterator?
Example76_BackInsertIteratorInside

```
1.template <class Container>  class back_insert_iterator {
2.protected:
3.  Container* container;
4.public:
5.typedef Container container_type;
6.explicit back_insert_iterator (Container& x) : container(&x) {}
7.// копирование значения
8.back_insert_iterator<Container>& operator= (const typename Container::value_type& value)
  {
9. container->push_back(value); return *this; }
10.// перемещение значения
11.back_insert_iterator<Container>& operator= (typename Container::value_type&& value)    {
12.container->push_back(std::move(value)); return *this; }
13.// стандартный набор операторов
14.back_insert_iterator<Container>& operator* ()    { return *this; }
15.back_insert_iterator<Container>& operator++ ()    { return *this; }
16.back_insert_iterator<Container> operator++ (int)    { return *this; }};
```

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ