Стандартная библиотека шаблонов

Введение. Контейнеры

Стандартная библиотека шаблонов (STL)

- Библиотека шаблонов является примером *обобщенного программирования*
- Содержит набор шаблонов (templates), представляющих *контейнеры, итераторы, функциональные объекты и алгоритмы*
- Реализует идеи шаблонов проектирования (patterns)
- Включена в стандарт C++ комитетом ISO/ANSI C++

Контейнеры

- Шаблоны классов для представления абстрактных структур данных
- Контейнеры представляют собой коллекции объектов
- Контейнеры в STL являются однородными, т.е. могут содержать только объекты одинакового типа
- В контейнерах для хранения элементов используется семантика передачи объектов по значению

Итераторы

- Итераторы – объекты, используемые для доступа к объектам в контейнере
- Итератор должен поддерживать понятие текущей позиции в контейнере
- Итератор должен предоставлять такие базовые операции, как продвижение к следующей позиции и обращение к элементу в текущей позиции
- Итераторы широко используют перегрузку операций, чтобы сделать работу с ними аналогичной работе с указателями

Шаблон класса vector

- Динамический массив однотипных объектов (аналог – массив)
- Произвольный доступ к элементами с автоматическим изменением размера при добавлении/удалении элемента
- Объявлен в заголовочном файле <vector>

Шаблон класса vector

- Доступ по индексу за $O(1)$
- Добавление-удаление элемента в конец за время $O(1)$ (если не изменяется размер)
- Добавление-удаление элемента в начале или середине $O(n)$
- Стандартная быстрая сортировка за $O(n * log_2(n))$
- Поиск элемента перебором занимает $O(n)$
- Существует специализация шаблона vector для типа bool, которая требует меньше памяти за счёт хранения элементов в виде битов, однако она не поддерживает всех возможностей работы с итераторами.

Пример использования

```
#include <vector>
using namespace std;
. . .
vector<int>  rating(5);
int n;
cin>>n;
vector<double> score(n);
vector<int> number;
```

```
// работа по аналогии с массивом
int i;
for (i=0; i<5; ++i) rating [ i ] = i;
for (i=0; i<n; ++i) cin>>score [ i ] ;
```

```
// используя методы контейнеров

int a;
for (i=0; i<5; ++i) {
   cin>>a;
    number.push_back(a);
}

cout<<number.size();
cout<<number.front()<<“ “<<number.back()<<endl;
number.pop_back();
```

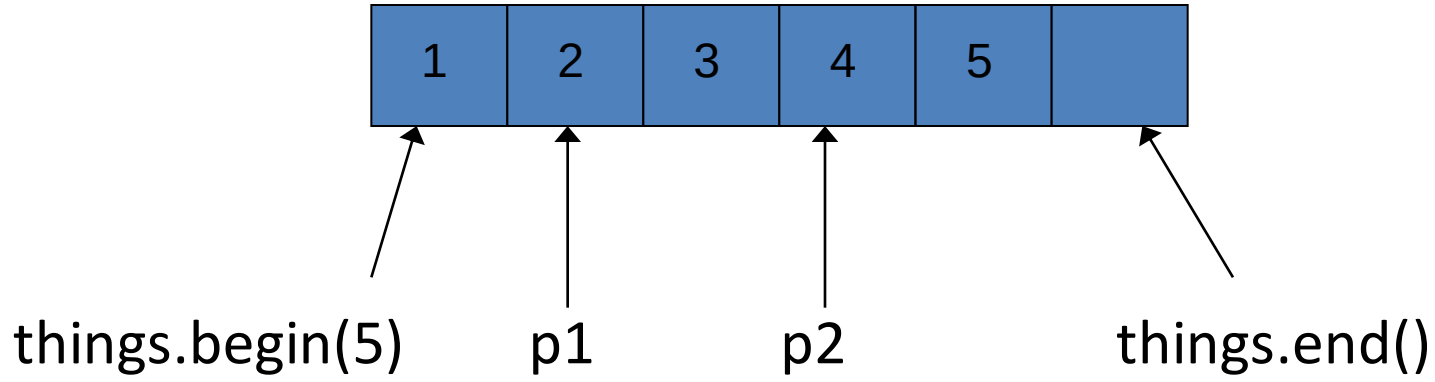
```
// используя методы контейнеров

vector <int> newarr(rating);

newarr.swap(number);

newarr.clear();

cout<<newarr.empty()<<endl;
```

Примеры использования итераторов

```
vector<double>:: iterator pd;
pd = score.begin();
*pd = 22.3;
++pd;
for (pd = score.begin();  pd!=score.end();  ++pd)
  cout<< *pd<< “  “;
```

Позиции итераторов и диапазоны

```
vector<int>  things(5);
```



Диапазон [things.begin() , things.end())

Диапазон [p1, p2)

Методы, работающие с указателями и диапазонами

a.erase(pt)	удаление элемента по указателю pt
a.erase(p1, p2)	удаление диапазона [p1, p2)

```
score.erase(score.begin(), score.begin()+2);
```

Методы, работающие с диапазонами

a.insert(p,t) вставка значения t перед указателем p

a.insert (pt, p1,p2) вставка диапазона [p1, p2) из другого вектора перед итератором pt

```
vector<int> old;
vector<int> new;
old.insert(old.begin(),3); old.insert(old.end(),5);
new.insert(new.begin(), old.begin(), old.end());
new.insert(new.end(), old.begin()+1, old.end()-1);
```

Обобщенные функции for_each()

```
MyFunc(MyType x);
vector<MyType> v;
vector<MyType>::iterator p;
for (p=v.begin(); p!= v.end(); ++p)
   MyFunc(*p);
. . .
for_each (v.begin(), v.end(), MyFunc);
```

MyFunc не должна изменять значения элементов контейнера

Обобщенные функции, переставляющие элементы

random_shuffle(a.begin(), a.end());

sort(a.begin(), a.end());

для такой сортировки требуется, чтобы для элементов, составляющих коллекцию была определена операция operator <()

sort(a.begin(), a.end(), WorseThan);

при сортировка используется функция-предикат WorseThan

```
struct student {
  string name;
  double r;
};
bool Rating (const student& s1,
                        const student& s2)
{  if (s1.r < s2.r) return true;
          else return false;
}
vector<student> sp;
sort (sp.begin(), sp.end(), Rating);
```

Дополнительно перегруженные операции

==

!=

<

>

<=

>=

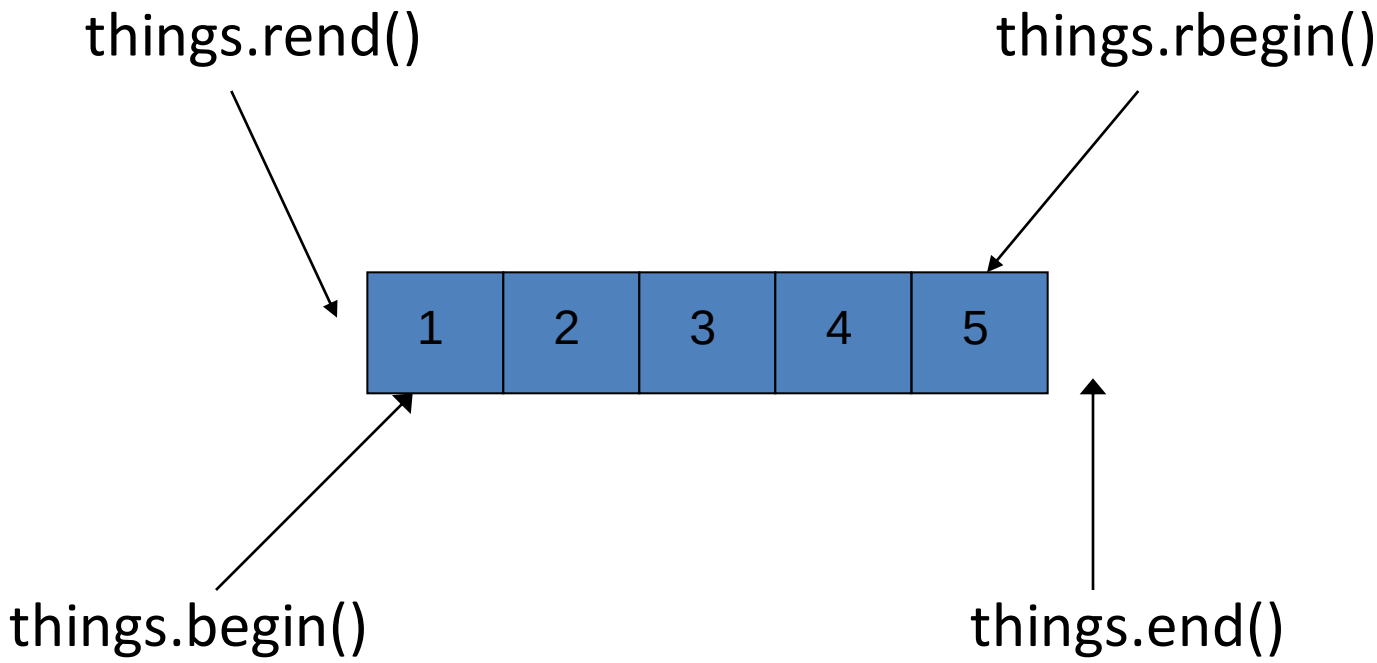
- Вектор – это контейнер, который является последовательным, т.е. организует хранение элементов в линейном порядке
- Вектор поддерживает итераторы произвольного доступа
- Операции вставки и удаления в конце вектора выполняются за постоянное время
- Операции вставки и удаления внутри вектора имеют линейную сложность
- Управление памятью выполняется автоматически

- Вектор является обратимым контейнером, что позволяет использовать с ним обратный итератор

```
vector<int> :: reverse_iterator rp;
vector<int> a;
rp=a.rbegin();
rp = a.rend();
```

Позиции итераторов

vector<int> things(5);



Пример

```
vector <int> v(n);

vector <int> ::  iterator pb;
for(pb=v.begin();  pb!=v.end(); ++pb) cout<<*pb<<" ";

vector <int> :: reverse_iterator rb;
for(rb=v.rbegin();  rb!=v.rend(); ++rb) cout<<*rb<<" "
```

Пример

```
pb=v.begin();
while(  pb!=v.end()  && *pb!=0) ++pb;

rb=v.rbegin();
while (  rb!=v.rend() && *rb!=0) ++rb;
```

Пример

```
vector <int>::iterator p;
//for (p=pb; p<rb ; ++p)                    ошибка
//       cout<<*p<<endl;

for (p=pb; p<rb.base(); ++p)
        cout<<*p<<“  “;
```

Типы контейнеров

- Стандартные
  - Последовательные (deque, list, vector)
  - Ассоциативные (map, multimap, set)
- Адаптеры (priority_queue, queue, stack)
- Псевдоконтейнеры (bitset, string, valarray, vector<bool>)

deque

- #include <deque>
- Двусвязный список массивов
- Подобно вектору поддерживает произвольный доступ к элементам
- Вставка и удаление возможны и в конце, и в начале очереди с постоянной сложностью $O(1)$
- Не безопасно использование арифметики указателей для доступа к элементам контейнера

list

- #include <list>
- Двусвязный список
- Обеспечивает вставку и удаление элементов в любом месте за постоянное время
- Произвольный доступ к элементам по индексу не возможен

Пример

```
list<int> l1(3,2); // 2 2 2
l1.push_back(10);  // 2 2 2 10
l1.push_front(5);  // 5 2 2 2 10
list<int> ::iterator pl;
pl=l1.begin();
++pl;
l1.insert(pl,0);        // 5 0 2 2 2 10
ostream_iterator<int,char> out(cout," ");
copy(l1.begin(),l1.end(),out);
```

Дополнительные методы

- remove (val)
- sort ( )

сложность $O(n * \log_2 n)$

- merge (other)

только для отсортированных

- unique ()

стирает все, кроме первого элемента, из каждой последовательной группы равных элементов в списке

- splice (pos, other)

вставляет содержимое *other* перед *pos*, и *other* становится пустым. Требуется постоянное время. Результат не определён, если *& other == this*.

Пример

```
l1.remove(2);// 5 0 10

int arr[5]={ 4,3,7,9,1};
l1.insert(l1.begin(),arr, arr+5);
//4 3 7 9 1 5 0 10
l1.sort();
// 0 1 3 4 5 7 9 10
```

Пример

```
list<int> l2;
int a2[10]={2,2,5,6,5,5,1,7,7,7};
l2.insert(l2.begin(),a2,a2+10);
// 2 2 5 6 5 5 1 7 7 7
l2.unique();
// 2 5 6 5 1 7
l2.sort();
// 1 2 5 5 6 7
l1.merge(l2);
// 0 1 1 2 3 4 5 5 5 6 7 7 9 10
```

Основные требования к элементам в контейнерах

Конструктор копии	Создаёт новый элемент, идентичный старому	Используется при каждой вставке элемента в контейнер
Оператор присваивания	Заменяет содержимое элемента копией исходного элемента	Используется при каждой модификации элемента
Деструктор	Разрушает элемент	Используется при каждом удалении элемента
Конструктор по умолчанию	Создаёт элемент без аргументов	Применяется только для определённых операций
operator==	Сравнивает два элемента	Используется при выполнении operator== для двух контейнеров
operator<	Определяет, меньше ли один элемент другого	Используется при выполнении operator< для двух контейнеров

выражение	**возвращаемый тип**	**семантика исполнения**	**контейнер**
a.front()	reference; const_reference для постоянного a	*a.begin()	vector, list, deque
a.back()	reference; const_reference для постоянного a	*a.(--end())	vector, list, deque
a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque
a.pop_front ()	void	a.erase(a.begin())	list, deque
a.pop_back ()	void	a.erase(-- a.end())	vector, list, deque
a[n]	reference; const_reference для постоянного a	*(a.begin() + n)	vector, deque

Адаптеры

- Устанавливают типичный интерфейс для определенных структур данных

queue

- Реализует интерфейс очереди
- Обычно реализуется через закрытое наследование от deque
- Не позволяет выполнять произвольный доступ к элементам, перемещаться по очереди
- Возможно добавление в конец очереди, удаление из начала очереди, определить размер очереди и проверить очередь на пустоту

Операции

- bool empty()
- size_type size()
- T& front()
- T& back()
- void push(const T&)
- void pop()

Пример

```
queue<int> q;
q.push (42);
q.push (101);
q.push (69);
while (!q.empty ())
 {
   cout << q.front () << endl;
   q.pop ();
 } // 42 101 69
```

priority_queue

- Набор операций совпадает с queue
- Наибольший элемент сдвигается в начало очереди
- Реализуется как вектор

Пример

```
priority_queue<int> q;

  q.push (42);
  q.push (101);
  q.push (69);
  while (!q.empty ())
  {
      cout << q.top() << endl;
    q.pop ();
  } // 101 69 42
```

stack

- Основан на классе вектор
- Интерфейс стека поддерживает операции
  - bool empty()
  - size_type size()
  - T& top()
  - void push(const T&)
  - void pop()

Пример

```
stack<int> q;


  q.push (42);
  q.push (101);
  q.push (69);
  while (!q.empty ())
  {
      cout << q.top() << endl;
    q.pop ();
  } // 69 101 42
```

Псевдоконтейнеры

- bitset
- Служит для хранения битовых масок.
- Похож на vector<bool> фиксированного размера. Размер фиксируется тогда, когда объявляется объект bitset. Итераторов в bitset нет. Оптимизирован по размеру памяти.

Псевдоконтейнеры

- basic_string
- Контейнер, предназначенный для хранения и обработки строк. Хранит в памяти элементы подряд единым блоком, что позволяет организовать быстрый доступ ко всей последовательности.
- Элементы должны быть POD.
- Определена конкатенация с помощью +.

Псевдоконтейнеры

- valarray
- Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности.
- В некоторой степени похож на vector, но в нём отсутствует большинство стандартных для контейнеров операций. Определены операции над двумя valarray и над valarray и скаляром (поэлементные).
- Эти операции возможно эффективно реализовать как на векторных процессорах, так и на скалярных процессорах с блоками SIMD.

Ассоциативные контейнеры

- Ассоциативные контейнеры обеспечивают быстрый поиск данных, основанных на ключах.
- Основные :
  - *set* (множество),
  - *multiset* (множество с дубликатами),
  - *map* (словарь) ,
  - *multimap*(словарь с дубликатами).

Ассоциативные контейнеры

- Ассоциативные контейнеры берут в качестве параметров *Key* (ключ) и упорядочивающее отношение *Compare*, которое вызывает полное упорядочение по элементам *Key*
- *set* и *multiset* хранят сами ключи
- *map* и *multimap* хранят пары <ключ,значение типа T>

set

- Функциональность:
  - добавить элемент в рассматриваемое множество, при этом исключая возможность появления дублей;
  - удалить элемент из множества;
  - узнать количество (различных) элементов в контейнере;
  - проверить, присутствует ли в контейнере некоторый элемент.

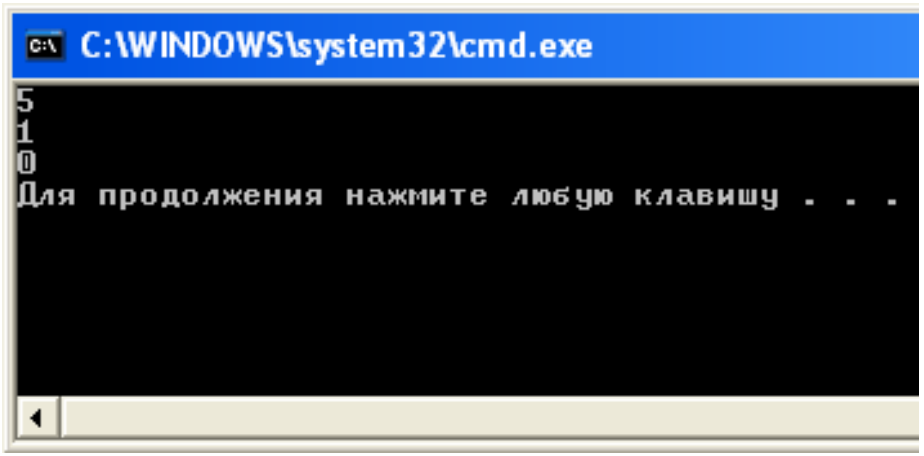
Пример

```
set<int> s;
for(int i = 1; i <= 100; i++)
    s.insert(i); // добавим 100 натуральных чисел
s.insert(42); // ничего не произойдёт
for(int i = 2; i <= 100; i += 2)
     s.remove(i); // удалим чётные числа
int N = int(s.size()); // N будет равно 50
int r = 0;
for(set<int>::const_iterator it = s.begin(); it != s.end(); ++it)
            r += (*it);
```

map

- хранит пары <key, value> упорядоченные по ключу
- операция индексирования перегружена как доступ к значению через ключ

```
map <char, int> M;
M.insert ( pair<char,int>('a',1) );
M['c'] = 5;
cout<< M['c']<<endl;
cout<< M['a']<<endl;
cout<< M['d']<<endl;
```



map

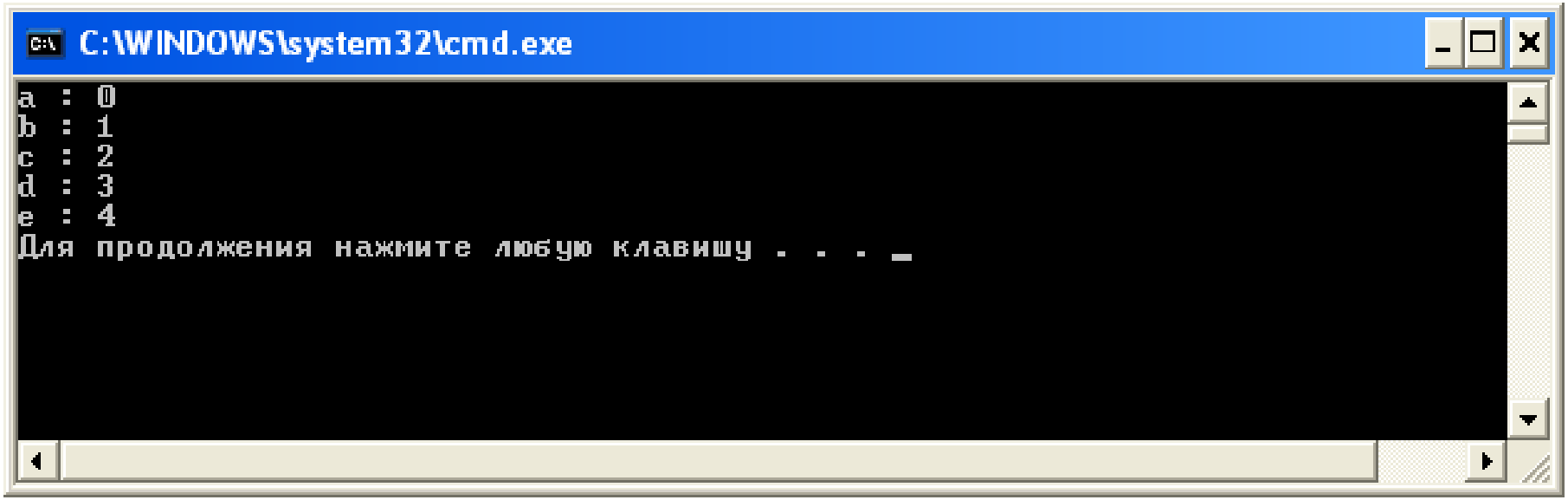
- если ключ не известен, то просмотреть элементы можно через итератор
- Следует помнить, что итератор указывает не на ключ key, а на пару <key, value>
- Чтобы получить значения из пары через итератор it используется

it->first или (*it).first

it->second или (*it).second

Пример

```
char c;
map <char,int> mySecondMap;
for (int i = 0,c = 'a'; i < 5; ++i,++c)
{
    mySecondMap.insert ( pair<char,int>(c,i) );
}

///вывод не явно инициализированной map на экран
for (auto it = mySecondMap.begin();
          it != mySecondMap.end(); ++it)
{
    cout << (*it).first << " : " << (*it).second
                 << endl;
}
```


C:\WINDOWS\system32\cmd.exe
a : 0
b : 1
c : 2
d : 3
e : 4
Для продолжения нажмите любую клавишу . . .