

# Стандартная библиотека шаблонов

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

---

- Контейнеры представляют собой коллекции объектов
- Контейнеры в STL являются однородными, т.е. могут содержать только объекты одинакового типа

# Итераторы

---

- Итераторы – объекты, используемые для обхода коллекции объектов
- 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`

---

- Коллекция однотипных объектов, к которым возможен произвольный доступ (аналог – массив)
- Объявлен в заголовочном файле `<vector>`

# 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 ] ;

// 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();  
vector <int> newarr(rating);  
newarr.swap(rating);  
newarr.clear();  
cout<<newarr.empty()<<endl;
```



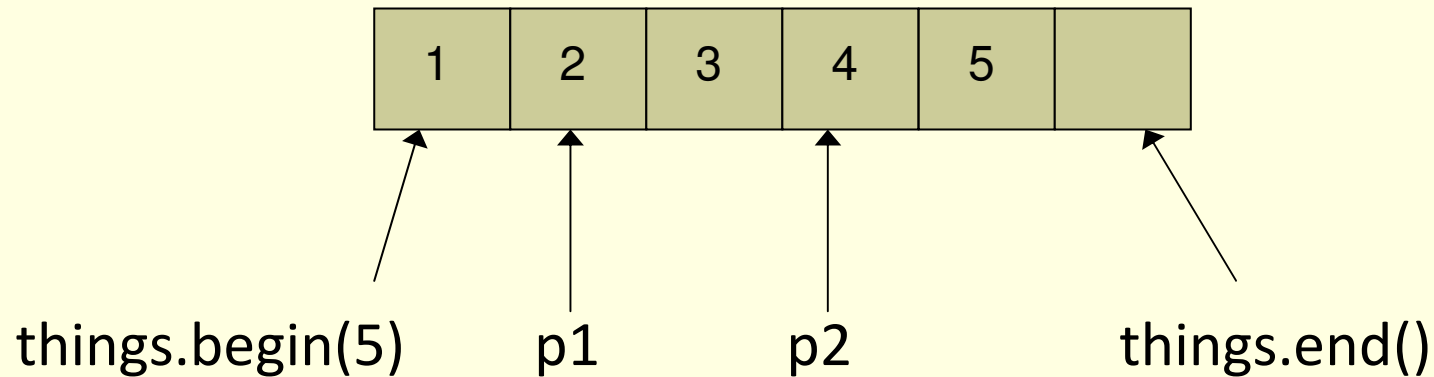
# Примеры использования итераторов

---

```
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 сортировке 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;
```

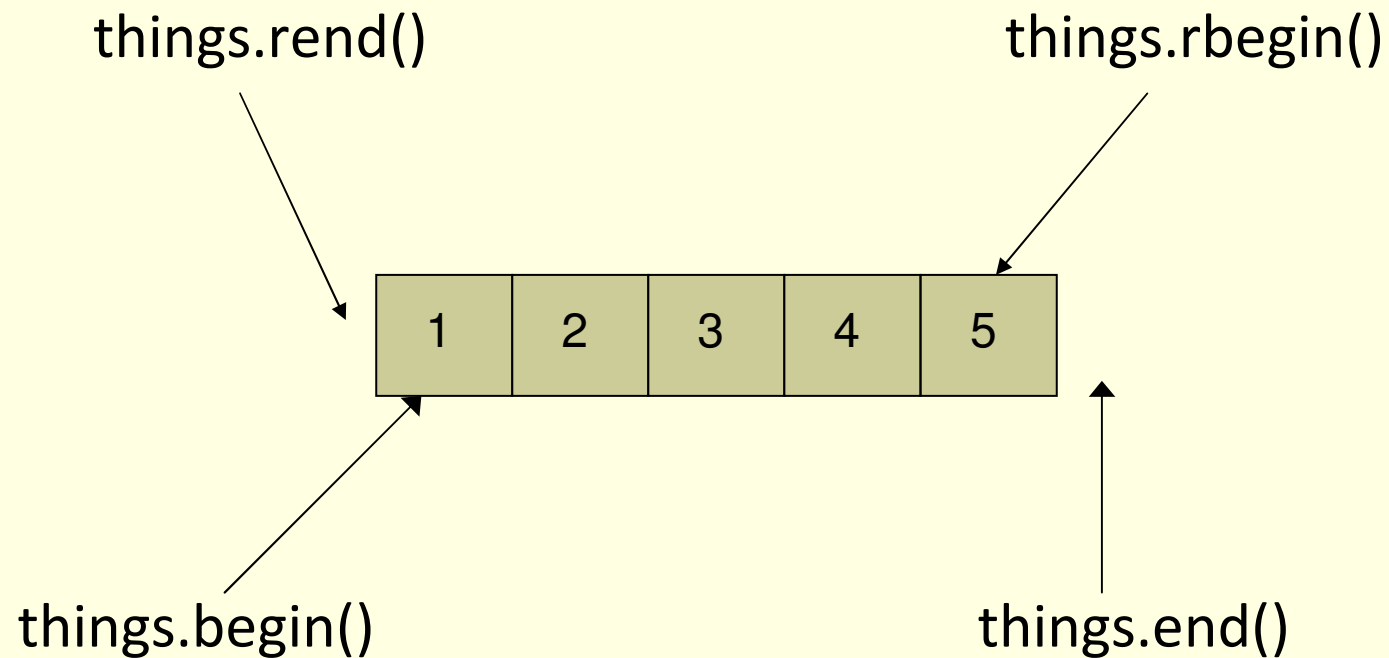
```
vector<int> a;
```

```
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)  
//    cout<<*p<<endl;
```

ошибка

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

# Пример

---

Чтобы исключить нули

```
++rb;  
for (p=pb+1; p<rb.base(); ++p)  
    cout<<*p<<endl;
```

# 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>`
- Двусторонняя очередь
- Подобно вектору поддерживает произвольный доступ к элементам
- Вставка и удаление возможны и в конце, и в начале очереди с постоянной сложностью

# list

---

- `#include <list>`
- Двусвязный список
- Обеспечивает вставку и удаление элементов в любом месте за постоянное время
- Не поддерживает произвольный доступ к элементам по индексу

# 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 p1;
p1=l1.begin();
++p1;
l1.insert(p1,0); // 5 0 2 2 2 10
ostream_iterator<int, char> out(cout, " ");
copy(l1.begin(), l1.end(), out);
```

# Дополнительные методы

---

- `remove ( val )`
- `sort ( )`                      сложность  $n \log n$
- `merge ( other)`
- `unique ( )`
- `splice ( pos, other)`

# 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
```

# Адаптеры

---

- Устанавливают интерфейс типичный 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
```