

Стандартная библиотека шаблонов

Итераторы

```
#include <iterator>
```

Требования к итераторам

- Наличие операции доступа к значению (разыменования)

$*p$

- Наличие операции присваивания

$p=q$

- Сравнение итераторов на равенство и неравенство

$p==q$ и $p!=q$

- Продвижение по всем элементам контейнера

$p++$ и $++p$

Связь итераторов и контейнеров

- Контейнер должен иметь методы, возвращающие позицию первого элемента контейнера

`begin()`

и позицию признака конца контейнера

`end()`

Основные свойства

- Итераторы всех типов можно разыменовывать (`*p`)
- Можно сравнивать на равенство и неравенство
- Если два итератора равны `p == q`, то и `*p == *q`

Типы итераторов

- Итераторы ввода (входные итераторы)
- Итераторы вывода (выходные итераторы)
- Итераторы перехода (последовательные итераторы)
- Двухнаправленные итераторы
- Итераторы произвольного доступа

Категории итераторов

Категория	Кем представлена
Входной	istream
Выходной	ostream, итераторы вставки
Прямой	Односвязный список
Двунаправленный	list, set, multiset, map, multimap
Произвольного доступа	vector, deque, basic_string, массивы

Итераторы ввода

- Предназначены для «ввода» информации из контейнера в алгоритм (программу)
- Позволяет прочесть значение из контейнера, но не позволяет изменить его (*p)
- Поддерживает операцию ++ в префиксной и постфиксной форме, проход по всем элементам контейнера один раз
- Операция присваивания трактуется как копирование
- Если значение итератора увеличено, то его нельзя разыменовывать по предыдущему значению, т.е. алгоритм должен быть однопроходным

Итераторы вывода

- Для записи данных из программы (алгоритма) в контейнер
- Нет операции присваивания
- Позволяет изменить значение в контейнере, но не позволяет прочесть его (*p=...)
- Только для однопроходных алгоритмов

Последовательные итераторы

- Обладают всеми свойствами входных и выходных операторов
- Могут указывать на один и тот же элемент в одной коллекции и обрабатывать его многократно
- Есть операция присваивания
- Если $r==s$, то $++r==++s$
- Могут использоваться в многопроходных однонаправленных алгоритмах

Двунаправленные итераторы

- Дополнительно вводятся операции
--р и р--

Итераторы произвольного доступа

- Нужны для алгоритмов, требующих перехода непосредственно к произвольному элементу массива (сортировка, двоичный поиск)
- Дополнительно определены операции

$p+n$

$p+=n$

$p-n$

$p-=n$

$p[n]$

$p-q$

$p<q$

$p>q$

$p<=q$

$p>=q$

Иерархия итераторов

- В алгоритмах, описанных для итератора определенного типа, может использоваться и любой другой итератор, имеющий требуемые возможности
- Алгоритмы разрабатываются так, чтобы требования к итератору были минимальными

- Для каждого контейнера определено итераторы какого типа он включает
- `vector <T>::iterator`
итератор произвольного доступа
- `list<t>:: iterator`
двунаправленный итератор
- Указатели удовлетворяют всем требованиям для итераторов, поэтому алгоритмы STL могут использоваться с массивами

Адаптеры итераторов

- Обратные итераторы

`reverse_iterator`

Переопределяют операции увеличения / уменьшения таким образом, что они действуют в обратном направлении

Существуют для двунаправленных итераторов и итераторов произвольного доступа

```
const int init[ ] = {1, 2, 3, 4, 5};  
vector<int> v(5);  
copy(init, init + 5, v.begin());  
copy(v.begin(), v.end(),  
      ostream_iterator<int>(cout, " "));  
  
copy(v.rbegin(), v.rend(),  
      ostream_iterator<int>(cout, " "));
```


Адаптеры итераторов

■ Итераторы вставки

Пусть `first`, `last` и `result` – обычные итераторы

```
while ( first != last )
```

```
    *result++ = *first++;
```

Требует, чтобы в `result` было достаточно элементов и замещает их

Итераторы вставки увеличивают контейнер

Переопределяют операции разыменования и инкремента как ничего не делающие

Адаптеры итераторов

■ Итераторы вставки

(в конструктор передается коллекция)

`back_insert_iterator`

`front_insert_iterator`

`insert_iterator` (и позиция)

Класс	Создание
<code>back_insert_iterator</code>	<code>back_inserter(coll)</code>
<code>front_insert_iterator</code>	<code>front_inserter(coll)</code>
<code>insert_iterator</code>	<code>inserter(coll, pos)</code>

Пример

```
char* array1 [] = { "laurie", "jennifer", "leisa" };
char* array2 [] = { "amanda", "saskia", "carrie" };
deque<char*> names (array1, array1 + 3);
insert_iterator<deque<char*>>
    it (names,names.begin()+2);
copy (array2, array2 + 3, it);
deque<char*>::iterator j;
for (j = names.begin (); j != names.end (); j++)
    cout << *j << endl;
```

Пример (2 способ)

```
char* array1 [] = { "laurie", "jennifer", "leisa" };  
char* array2 [] = { "amanda", "saskia", "carrie" };  
deque<char*> names (array1, array1 + 3);
```

```
copy (array2, array2 + 3,  
      insert_iterator<deque<char*>>  
      (names,names.begin()+2));
```

```
deque<char*>::iterator j;  
for (j = names.begin (); j != names.end (); j++)  
    cout << *j << endl;
```

Пример (3 способ)

```
char* array1 [] = { "laurie", "jennifer", "leisa" };  
char* array2 [] = { "amanda", "saskia", "carrie" };  
deque<char*> names (array1, array1 + 3);
```

```
deque<char*>::iterator i = names.begin () + 2;  
copy (array2, array2 + 3, inserter (names, i));
```

```
deque<char*>::iterator j;  
for (j = names.begin (); j != names.end (); j++)  
cout << *j << endl;
```

Итераторы входного/выходного ПОТОКОВ

`istream_iterator`

Для адаптации потока ввода к интерфейсу
итераторов

`ostream_iterator`

Адаптер потока вывода к интерфейсу
итераторов

Пример

```
deque <int> s;  
copy(istream_iterator<int, char>(cin),  
     istream_iterator<int, char>(),  
     back_inserter(s));  
copy(s.begin(),  
     s.end(),  
     ostream_iterator<int, char>(cout, "---"));
```

Ввод:

1 2 3 4

5 6.7 8

Вывод

1---2---3---4---5---6---