Стандартная библиотека STL: итераторы и функторы

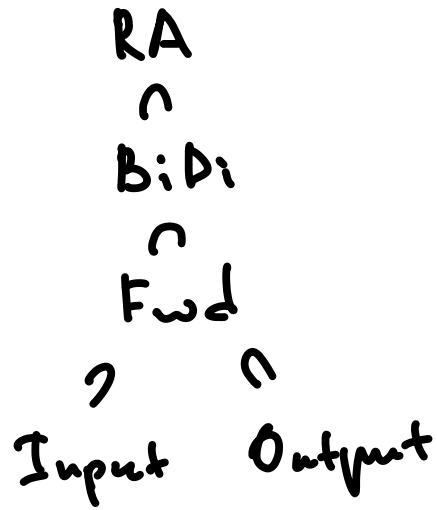
Александр Смаль

Академический университет
27 февраля 2014
Санкт-Петербург

Категории итераторов

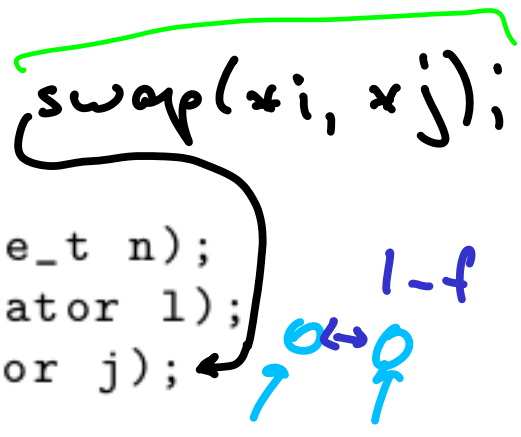
Итератор — синтаксически похожий на указатель объект для доступа к элементам последовательности.
Итераторы делятся на пять категорий.

- Random access iterator (произвольного доступа)
  ++,--, арифметика, read-write
- Bidirectional iterator (двунаправленный)
  ++,--, read-write
- Forward iterator (однонаправленный)
  ++, read-write
- Input iterator (итератор ввода)
  ++, read
- Output iterator (итератор вывода)
  ++, write



Функции для работы с итераторами:

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



iterator_traits

```
// <iterator>

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

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

size_t

T

T*

T&

Iterator2::value_type
не работает если
Iterator = V*

iterator_traits для указателей

namespace std {
template <>
void swap<MyT>(MyT & a, MyT & b) {}
}

```
// <iterator>

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

iterator_category

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

A a = A();

string
string a;

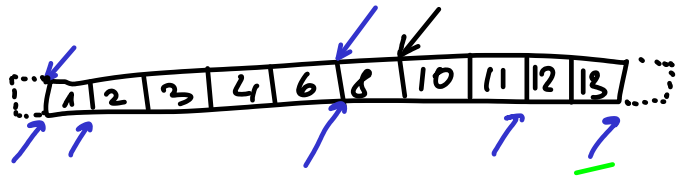
```
template<class Iterator>
void advance(Iterator & i, size_t n) {
    advance_impl(i, n, typename iterator_traits<Iterator>::
                        iterator_category());
}
template<class Iterator>
void advance_impl(Iterator & i, size_t n,
                  random_advance_iterator_tag)
{ i += n; }

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:

```
reverse_iterator rbegin();
reverse_iterator rend();
```




```
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;
```

typedef std::reverse_iterator<ID> reverse_iterator;

Advanced итераторы

В <iterator> определены следующие специальные итераторы итераторов:

1. back_inserter, front_inserter, inserter
2. istream_iterator, ostream_iterator

A a(B());

```
vector<Person> db;
template<class OutIt>
void findByName(vector<Person> const& db,
                string name, OutIt out);
...
vector<Person> res; // size is not known
findByName(db, "Ivan", std::back_inserter(res));
```

```
ifstream file("input.txt");
vector<double> v((istream_iterator<double>(file)),
                 istream_iterator<double>());
copy(v.begin(), v.end(),
     ostream_iterator<double>(cout, '\n'));
```

(II, (File), II())

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>
{
// ++, --, ...
};
```

Функторы и предикаты

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`
- *Предикат* — функтор возвращающий `bool`.

Функторы в стандартной библиотеке:

1. `less`, `greater`, `less_equal`, `greater_equal`, `not_equal_to`, `equal_to`
2. `minus`, `plus`, `divides`, `modulus`, `multiplies`
3. `logical_not`, `logical_and`, `logical_or`
4. `mem_fun`, `mem_fun_ref`, `ptr_fun`
5. `bind1st`, `bind2nd`, `not1`, `not2`

```
map<int, string, greater<int>> m;

bind2nd(less<int>(), 10);
bind1st(modulus<size_t>, 1024);
not1(logical_not<bool>());
```

Как написать свой функтор

```
#include <functional>

template <class Arg, class Result>
  struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
  };

template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
  };

struct MyFunctor :
    binary_function<int, double, size_t>
{
    size_t operator()(int i, double d) {...}
};
```

boost::bind

~~::lambda::bind~~