Программирование на языке C++
Лекция 9

Итераторы и умные указатели

Александр Смаль

Категории итераторов

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

- Random access iterator: `++`, `--`, арифметика, `<`, `>`, `<=`, `>=`. (`array`, `vector`, `deque`)
- Bidirectional iterator: `++`, `--`. (`list`, `set`, `map`)
- Forward iterator: ++. (`forward_list`, `unordered_set`, `unordered_map`)
- Input iterator: ++, read-only.
- Output iterator: ++, write-only.

Функции для работы с итераторами:

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

iterator_traits

```
// заголовочный файл <iterator>
template <class Iterator>
struct iterator_traits {
   typedef difference_type   Iterator::difference_type;
   typedef value_type        Iterator::value_type;
   typedef pointer           Iterator::pointer;
   typedef reference         Iterator::reference;
   typedef iterator_category Iterator::iterator_category;
};

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

```
// <iterator>
struct random_access_iterator_tag {};
struct bidirectional_iterator_tag {};
struct forward_iterator_tag {};
struct input_iterator_tag {};
struct output_iterator_tag {};
```

```
template<class I>
void advance(I & i, size_t n,
             random_access_iterator_tag)
{ i += n; }
template<class I>
void advance(I & i, size_t n, ... ) {
   for (size_t k = 0; k != n; ++k, ++i );
}
template<class I>
void advance(I & i, size_t n) {
   advance(i, n, typename
     iterator_traits<I>::iterator_category());
}
```

reverse_iterator

У некоторых контейнеров есть обратные итераторы:

```
list<int> l = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// list<int>::reverse_iterator
for(auto i = l.rbegin(); i != l.rend(); ++i)
    cout << *i << endl;
```

Конвертация итераторов:

```
list<int>::iterator i = l.begin();
advance(i, 5); // i указывает на 5
// ri указывает на 4
list<int>::reverse_iterator ri = i;
i = ri.base();
```

Есть возможность сделать обратный итератор из random access или bidirectional при помощи шаблона `reverse_iterator`.

```
// <iterator>
template <class Iterator>
class reverse_iterator {...};
```

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.
3. В `vector` и `string` удаление элемента инвалидирует итераторы на все следующие элементы.

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.
3. В `vector` и `string` удаление элемента инвалидирует итераторы на все следующие элементы.
4. В `deque` удаление/добавление инвалидирует все итераторы, кроме случаев удаления/добавления первого или последнего элементов.

Advanced итераторы

Для пополнения контейнеров:
`back_inserter, front_inserter, inserter`.

```
// в классе Database
    template<class OutIt>
    void findByName(string name, OutIt out);
```

```
// размер заранее неизвестен
vector<Person> res;
Database::findByName("Rick", back_inserter(res));
```

Для работы с потоками:
`istream_iterator, ostream_iterator`.

```
ifstream file("input.txt");
vector<double> v((istream_iterator<double>(file)),
                 istream_iterator<double>());

copy(v.begin(), v.end(),
     ostream_iterator<double>(cout, '\n'));
```

Как написать свой итератор

```
// <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;
```

```
#include <iterator>

struct PersonIterator
    : std::iterator<forward_iterator_tag, Person>
{
// operator++, operator*, ...
};
```

Умные указатели

`unique_ptr`

- Умный указатель с уникальным владением.
- Нельзя копировать, можно перемещать.
- Не подходит для разделяемых объектов.

`shared_ptr`

- Умный указатель с подсчётом ссылок.
- Универсальный указатель.

`weak_ptr`

- Умный указатель с для создания *слабых ссылок*.
- Работает вместе с `shared_ptr`.