Лекция 9. Итераторы. Исключения.

Илья Макаров

ИТМО JB

2 ноября 2021

Санкт-Петербург

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`)
- C++20: **Contiguous iterator**: `++`, `--`, арифметика, `<`, `>`, `<=`, `>=`. (`array`, `vector`)
- **Bidirectional iterator**: `++`, `--`. (`list`, `set`, `map`)
- **Forward iterator**: `++`. (`forward_list`, `unordered_set`, `unordered_map`)
- **Input iterator**: ++, read-only.
- **Output iterator**: ++, write-only.

Работа с итераторами

Функции для работы с итераторами:

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

iterator_traits

Позволяет реализовывать алгоритмы в терминах итераторов, даже если тип не имеет соответствующих `typedef`.

```
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;
};
```

iterator_traits

```
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_traits

```
template<class BidirIt>
void my_reverse(BidirIt first, BidirIt last)
{
    using diff_t = typename
        std::iterator_traits<BidirIt>::difference_type;
    diff_t n = std::distance(first, last);
    for (--n; n > 0; n -= 2) {
        uisng value_t = typename
            std::iterator_traits<BidirIt>::value_type;
        const value_t tmp = *first;
        *first++ = *--last;
        *last = tmp;
    }
}
```

iterator_category

Позволяют выбрать более оптимальную реализацию алгоритма.

```
struct random_access_iterator_tag {};
struct bidirectional_iterator_tag {};
struct forward_iterator_tag {};
struct input_iterator_tag {};
struct output_iterator_tag {};
struct contiguous_iterator_tag {}; // C++20
```

iterator_category

```
template<class I>
void advance_impl(I & i, long n, random_access_iterator_ta
    i += n;
}

template<class I>
void advance_impl(I & i, size_t n, ... ) {
   for (size_t k = 0; k != n; ++k, ++i );
}

template<class I>
void advance(I & i, size_t n) {
    using it_cat = typename
        iterator_traits<I>::iterator_category
   advance_impl(i, n, it_cat());
}
```

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

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).

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));
```

Advanced итераторы

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

```
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
>
struct iterator;
```

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

Логические ошибки и исключительные ситуации

- **Логические ошибки**.
  Ошибки в логике работы программы, которые происходят из-за неправильно написанного кода, т.е. это ошибки программиста:
  - выход за границу массива,
  - попытка деления на ноль,
  - обращение по нулевому указателю,
  - …
- **Исключительные ситуации**.
  Ситуации, которые требуют особой обработки. Возникновение таких ситуаций — это „нормальное“ поведение программы.
  - ошибка записи на диск,
  - недоступность сервера,
  - неправильный формат файла,
  - …

Выявление логических ошибок на этапе разработки

Оператор `static_assert`.

```
#include<type_traits>

template<class T>
void countdown(T start)
{
    static_assert(std::is_integral<T>::value
                  && std::is_signed<T>::value,
                     "Requires signed integral type");

    while (start >= 0) {
        std::cout << start-- << std::endl;
    }
}
```

Выявление логических ошибок на этапе разработки

Макрос `assert`.

```
#include<type_traits>
//#define NDEBUG
#include <cassert>

template<class T>
void countdown(T start)
{
    assert(start >= 0);
    while (start >= 0) {
        std::cout << start-- << std::endl;
    }
}
```

Способы сообщения об ошибке

Возврат статуса операции:

```
bool write(string file, string data, size_t & bytes);
```

Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

Глобальная переменная для кода ошибки:

```
size_t write(string file, string data);
```

```
size_t bytes = write(f, data);
if (errno) {
    cerr << strerror(errno);
    errno = 0;
}
```

Исключения

```
size_t write(string file, string data) {
    if (!open(file)) throw FileOpenError(file);
    //...
}

double safediv(int x, int y) {
    if (y == 0) throw MathError("Division by zero");
    return double(x) / y;
}
```

Исключения

```
void write_x_div_y(string file, int x, int y) {
    try {
        write(file, to_string(safediv(x, y)));
    } catch (MathError & s) {
        // обработка ошибки в safediv
    } catch (FileError & e) {
        // обработка ошибки в write
    } catch (...) {
        // все остальные ошибки
    }
}
```

Stack unwinding

При возникновении исключения объекты на стеке уничтожаются в естественном (обратном) порядке.

```
void foo() {
    A a;
    if (!a) throw Error();
    B b;
}
void bar() {
    C d;
    try {
        D d;
        foo();
    } catch (const Error &) { throw OtherError; }
}
```

Почему не стоит бросать встроенные типы

```
int foo() {
    if (...) throw -1;
    if (...) throw 3.1415;
}
void bar(int a) {
    if (a == 0) throw string("Not my fault!");
}
int main () {
    try { bar(foo()); }
    catch (string & s) { /*only str*/ }
    catch (int a) { /*only int*/ }
    catch (double d) { /*only double*/ }
    catch (...) { /*nothing*/ }
}
```

Стандартные классы исключений

Базовый класс для всех исключений (в <exception>):

```
struct exception {
  virtual ~exception();
  virtual const char* what() const;
};
```

Стандартные классы ошибок (в stdexcept):

- logic_error: domain_error, invalid_argument, length_error, out_of_range
- runtime_error: range_error, overflow_error, underflow_error

Исключения в стандартной библиотеке

- Метод `at` контейнеров `array`, `vector`, `deque`, `basic_string`, `bitset`, `map`, `unordered_map` бросает `out_of_range`.
- Оператор `new T` бросает `bad_alloc`.
  Оператор `new (std::nothrow) T` в возвращает 0.
- Оператор `typeid` от разыменованного нулевого указателя бросает `bad_typeid`.

Исключения в стандартной библиотеке

- Потоки ввода-вывода.

```
std::ifstream file;
file.exceptions(std::ifstream::failbit
    | std::ifstream::badbit);
try {
    file.open("test.txt");
    cout << file.get() << endl;
    file.close();
}
catch (std::ifstream::failure const& e) {
    cerr << e.what() << endl;
}
```

Исключения в деструкторах

Исключения не должны покидать деструкторы.

- Двойное исключение:

```
void foo() {
    try {
        Bad b; // исключение в деструкторе
        bar(); // исключение
    } catch (std::exception & e) {
        // ...
    }
}
```

- Неопределённое поведение:

```
void bar() {
    Bad * bad = new Bad[100];
    delete [] bad; // ислючение в деструкторе
}
```

Исключения в конструкторе

Исключения — это единственный способ прервать конструирование объекта и сообщить об ошибке.

```
struct Database {
   explicit Database(string const& uri) {
      if (!connect(uri)) throw ConnectionError(uri);
   }
   ~Database() { disconnect(); }
};
int main() try {
    Database * db = new Database("db.local");
    db->dump("db-local-dump.sql");
    delete db;
} catch (std::exception const& e) {
    std::cerr << e.what() << std::endl;
}
```

Исключения в списке инициализации

Позволяет отловить исключения при создании полей класса.

```
struct System
{
    System(string const& uri, string const& data)
    try : db_(uri), dh_(data)
    { ... }
    catch (std::exception & e)
    {
        log("System constructor: ", e);
        throw;
    }

    Database    db_;
    DataHolder  dh_;
};
```

Как обрабатывать ошибки?

Есть несколько „правил хорошего тона“.

- Разделяйте ошибки программиста и исключительные ситуации.
- Используйте `assert` и `static_assert` для выявления ошибок на этапе разработки.
- В пределах одной логической части кода обрабатывайте ошибки централизованно и однообразно.
- Обрабатывайте ошибки там, где их можно обработать.
- Если в данном месте ошибку не обработать, то пересылайте её выше при помощи исключения.
- Бросайте только стандартные классы исключений или производные от них.
- Бросайте исключения по значению, а ловите по ссылке.
- Отлавливайте все исключения в точке входа.