

# Программирование на C++. Часть 2

## Лекция 11

ПМИ Семестр 2

Демяненко Я.М.

2026

# Итераторы ввода/вывода

Важной причиной включения итераторов ввода/вывода в STL является возможность **выделения алгоритмов**, которые могут **использоваться с итераторами, связанными с потоками ввода-вывода**.

Такие итераторы предоставляются классами STL **istream\_iterator** (для ввода) и **ostream\_iterator** (для вывода).

# Итераторы ввода — 1

Итераторы ввода предназначены для получения информации **из последовательности в алгоритм**, при этом значение в последовательности **не может быть изменено**.

Под последовательностью в данном случае понимаются или контейнер или поток ввода.

Итераторы ввода поддерживают операцию ++ и используются **в однопроходных алгоритмах**.

# Итераторы ввода — 2

Для входных итераторов из  $a == b$  не следует  $++a == ++b$

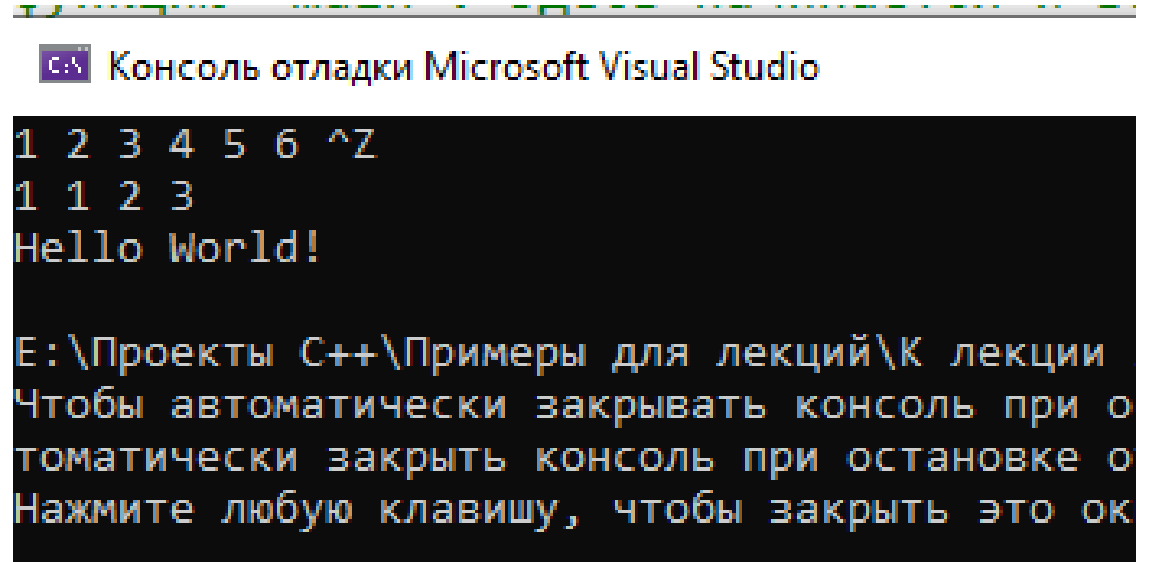
Допустим      1 2 3 4 5 6  
`istream_iterator<char> a(cin);` — указывает на 1  
`istream_iterator<char> b=a;` — указывает на 1  
`++a;` — указывает на 2  
`++b;` — указывает на 3

Алгоритмы, работающие с входными итераторами, не должны пытаться скопировать значение итератора и использовать его для повторного прохода по одной и той же позиции.

Тип значения, возвращаемого итератором, не обязан быть ссылочным, поскольку алгоритмы, работающие со входными итераторами, не должны пытаться посредством них выполнять присваивания.

# Итераторы ввода — 3

```
int main() {  
    istream_iterator<char> a(cin);  
    istream_iterator<char> b=a;  
    auto x1 = *a;  
    auto x2 = *b;  
    ++a;  
    auto x3 = *a;  
    ++b;  
    auto x4 = *b;  
    cout << x1 << ' ' << x2 << ' ' << x3 << ' ' << x4 << endl;  
    std::cout << "Hello World!\n";  
}
```



Консоль отладки Microsoft Visual Studio

```
1 2 3 4 5 6 ^Z  
1 1 2 3  
Hello World!  
  
E:\Проекты С++\Примеры для лекций\К лекции  
Чтобы автоматически закрывать консоль при о  
томатически закрыть консоль при остановке о  
Нажмите любую клавишу, чтобы закрыть это ок
```

# Итераторы ввода — 4

```
int main() {  
    istream_iterator<char> a(cin);  
    auto x1=*a;  
    istream_iterator<char> b(cin);  
    auto x2 = *b;  
    ++a;  
    auto x3 = *a;  
    ++b;  
    auto x4 = *b;  
    cout << x1 << ' ' << x2 << ' ' << x3 << ' ' << x4 << endl;  
    std::cout << "Hello World!\n";  
}
```

Консоль отладки Microsoft Visual Studio

```
1 2 3 4 5 6^Z  
1 2 3 4  
Hello World!  
  
E:\Проекты C++\Примеры для лекций\К лекции 2_11\  
Чтобы автоматически закрывать консоль при остано  
томатически закрыть консоль при остановке отладк  
Нажмите любую клавишу, чтобы закрыть это окно...
```

**Пример.** Выполнить поиск первого вхождения заданного значения в массиве, списке и потоке ввода `istream`. Вывести для каждой коллекции или элемент, следующий за найденным, или сообщение, что найденный — последний, или сообщение, что элемент не найден.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main() {
int a[10] = { 12, 3, 25, 7, 11, 213, 7, 123, 29, -3 };    // Инициализация массива 10 целыми числами

int* ptr = find(a, a+10, 7); // Поиск в массиве первого элемента, равного 7

if (ptr < a+10 )
    if ((++ptr) < a+10) cout << *(ptr) << endl;
    else cout << "Last found\n";
else cout << "Not found\n";
```

```

list<int> listl(a+4, a+10); // Инициализация списка числами из массива, начиная с a[4]

auto pi = find(begin(listl), end(listl), 7); // Поиск в списке первого элемента, равного 7

if (pi !=end(listl))
    if ((++pi) !=end(listl)) cout << *(pi) << endl;
    else cout << "Last found\n";
else cout << "Not found\n";

istream_iterator<char> in(cin);
istream_iterator<char> eos;
in = find(in, eos, '7'); // Поиск первого символа во входном потоке, равного 7

if ((in) != eos)
    if ((++in) != eos)
        cout << *(in) << endl;
    else cout << "Last found\n";
else cout << "Not found\n";
}

```

# Итераторы вывода — 1

Выходные итераторы (итераторы вывода) обладают **противоположной входным** итераторам функциональностью. Они позволяют **записывать значения в последовательность**.

Под последовательностью понимается либо контейнер, либо выходной поток.

Как и в случае итераторов ввода, **алгоритмы**, использующие итераторы вывода, должны быть **однопроходными**.

# Итераторы вывода — 2

Операции равенства и неравенства для таких итераторов могут быть не определены.

Алгоритмы, использующие итераторы вывода, могут работать с выходными потоками для размещения данных посредством класса `ostream_iterator`, а также с итераторами и указателями вставки.

Единственное корректное применение операции `operator*` к итераторам вывода — в левой части операции присваивания.

Выходной итератор в алгоритме `copy` для вывода символов в поток `cout`

```
copy(text.begin(),text.end(),ostream_iterator<char>(cout,""));
```

# Однонаправленный итератор

Это итератор, который **объединяет** свойства **входного и выходного** итераторов, тем самым обеспечивая возможность чтения и записи.

При этом обход элементов контейнера происходит в одном направлении.

**Однонаправленные** итераторы обладают также возможностью **сохранить** значение итератора и использовать сохраненное значение **для повторного прохода** из той же позиции.

Это позволяет использовать однонаправленные итераторы в **многопроходных** алгоритмах.

# Однонаправленные и двунаправленные

Контейнер **forward\_list** имеет **однонаправленный** итератор **forward\_list<T>::iterator**.

Последовательные контейнеры **vector**, **list**, **deque**, а также **массивы**, имеют **двунаправленные** итераторы, которые **обладают свойствами однонаправленных**.

Поэтому к ним тоже применимы алгоритмы, использующие однонаправленные итераторы.

# Алгоритм replace

Заменяет в диапазоне все элементы с заданным значением другим значением

Ему необходим однонаправленный итератор

```
int a[10];
for (int i=0; i<10; ++i)
    cin >> a[i];
// Замена всех элементов массива, равных
5, на 6
replace(a, a+10, 5, 6);
```

```
deque<char> deq;
char c;
for (int i=0; i<10; ++i) {
    cin >> c;
    deq.push_back(c);
    deq.push_front(c);
}
// Замена всех 'e' в deq на 'o'
replace(deq.begin(), deq.end(), 'e', 'o');
```

Поскольку тип `deque<T>::iterator` также удовлетворяет всем требованиям к однонаправленным итераторам, этот алгоритм можно использовать и для замены элементов в деке

# Двунаправленный итератор

Аналогичен однонаправленному, кроме того, он допускает обход в обоих направлениях.

То есть, двунаправленные итераторы должны поддерживать все операции однонаправленных итераторов, а кроме них — **операцию** `--`, делая возможным обход последовательности в обратном направлении.

И префиксный, и постфиксный `operator--` должны выполняться за константное время.

Возможность обхода структуры данных в обратном порядке важна потому, что без неё некоторые алгоритмы не могут эффективно работать.

Например, **алгоритм** STL `reverse` может использоваться для обращения порядка элементов в последовательности только при наличии **двунаправленных** итераторов.

# Контейнер `list<T>`

Предоставляет двунаправленные итераторы, так что его можно использовать с алгоритмом **reverse**

```
list<int> list1;  
// Код для вставки значений в list1  
reverse(begin(list1),end(list1)); // Обращение порядка значений в списке
```

Возможность предоставления контейнером двунаправленного итератора зависит от внутренней организации контейнера.

Контейнер `list<T>` реализован как двусвязный список, поэтому предоставляет возможность итератору реализовать операцию `operator--` за константное время.

В случае односвязного списка эффективно реализовать `operator--` (с константным временем работы) невозможно.

# Достижимость элементов

Имеются некоторые алгоритмы, для **эффективной** работы которых требуется, чтобы **любой** элемент последовательности был **достижим** из любого другого **за константное время**.

Например, обобщённый алгоритм бинарного поиска

```
binary_search(first, last, value);
```

Для **эффективной** работы таких алгоритмов **необходимы** итераторы **произвольного** доступа.

# Итераторы произвольного доступа

Должны поддерживать все операции двунаправленных итераторов, плюс нижеперечисленные

- Прибавление и вычитание целых чисел:  $r+n$ ,  $n+r$ ,  $r-n$ ,  $r+=n$  и  $r-=n$
- Доступ к  $n$ -му элементу при помощи выражения  $r[n]$ , которое означает  $*(r+n)$
- Вычитание итераторов  $r-s$
- Сравнения  $r<s$ ,  $r>s$ ,  $r<=s$  и  $r>=s$

# Алгоритм `binary_search`

Алгоритм `binary_search` на таких контейнерах как **векторы, деки и массивы**, имеет наибольшую эффективность — **время работы  $O(\log N)$** , т.к. их итераторы являются итераторами **произвольного доступа**.

Однако **требование** алгоритма — наличие **однонаправленного** итератора, позволяет использовать его и для других контейнеров, например, списков.

Но время работы увеличится, т.к. имея указатели на начало и конец списка, единственный способ добраться до элемента посередине — это обойти элементы один за другим.

Это займет время, пропорциональное длине списка.

# Адаптеры итераторов

**Адаптеры** — шаблонные классы, которые обеспечивают изменение поведения при сохранении интерфейса.

Например, для двунаправленных итераторов и итераторов произвольного доступа существуют адаптеры — **обратные (реверсивные) итераторы** — `reverse_iterator`.

Они переопределяют операции увеличения (уменьшения) таким образом, что перемещение происходит в противоположном направлении.

Для инициализации реверсивных итераторов используются операции `rbegin()` и `rend()`.

# Проверка контейнера на симметричность

```
auto itb = v.begin();
auto itr = v.rbegin();
while (itb && itb < itr.base()) {
    b = (*itb) == (*itr);
    itb++;
    itr++;
}
```

# Порядок сортировки

Например, чтобы **изменить порядок сортировки** вектора с возрастания на убывание, можно **вместо** написания своего **компаратора** использовать **реверсивные итераторы**.

```
// сортирует вектор в порядке возрастания  
sort( v.begin(), v.end() );  
// сортирует вектор в порядке убывания  
sort( v.rbegin(), v.rend() );
```

## И ещё адаптеры итераторов — итераторы вставки

Главное отличие этих итераторов состоит в том, что выражение

`*it = value`

трактруется как вставка в контейнер нового элемента со значением `value`.

Позиция вставки при этом определяется типом итератора `it`.

# Три типа итераторов вставки

Существует три типа итераторов вставки, параметризованные типом контейнера:

- `insert_iterator<Container>` – использует функцию `insert()` класса `Container`
- `back_insert_iterator<Container>` – использует функцию `push_back()`
- `front_insert_iterator<Container>` – использует функцию `push_front()`

# Объявление итераторов вставки

Например, если  $v$  — вектор целых чисел, то объявление для него итератора вставки перед вторым элементом будет выглядеть следующим образом.

```
insert_iterator<vector<int>> it(v,next(begin(v)));
```

# Шаблоны функций, возвращающие итераторы

Чтобы было удобнее использовать вводятся шаблоны функций, возвращающие нужные итераторы

- `back_inserter`
- `front_inserter`
- `inserter`

Так для нашего примера объявление будет выглядеть следующим образом

```
auto it = inserter(v, next(begin(v)));
```

# Итераторы вставки

```
vector<int> v(10); //123
vector<int> v1;
copy(begin(v), end(v), back_inserter(v1));
//123
```

```
vector<int> v(10); //123
list<int> l1;
copy(begin(v), end(v), front_inserter(l1));
//321
```

```
vector<int> v(10); //123
vector<int> v1; //123
copy(v.begin(), v.end(), inserter(v1, next(begin(v1))));
//112323
```

# Примеры использования insert

```
vector <int> vecInt(3,100);          ///Создаем вектор из 3 элементов и заполняем его  
                                  ///значением 100
```

```
vector <int>::iterator it;  
it = begin(vecInt);                ///Итератор указывает на vec[0]
```

```
                                  ///Вектор расширяется теперь до 4 элементов  
vecInt.insert(it,200);             ///И первым элементом записывается 200
```

```
                                  ///Вектор расширяется теперь до 5 элементов  
it = vecInt.begin() + 3;          ///Итератор указывает на 4 элемент (0-элемент+3-элемента)  
vecInt.insert(it,300);            ///И четвертым элементом записывается 300
```

```
                                  ///Вектор расширяется теперь до 6 элементов  
vecInt.insert(it+1,900);          ///И 5 элементом записывается 900
```

# Рекомендации по выводу и вводу

```
vector<int> v(10);  
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

```
vector<int> v;  
copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
```

# Как устроено внутри

```
// Допустим нужно копировать последовательность vector<int> v{1, 5, 3}; в поток cout  
// хочется написать так copy(begin(v), end(v), cout)
```

```
// На деле создаётся обертка над cout
```

```
copy(begin(v), end(v), ostream_iterator<int>(cout, " "));
```

```
template<typename T>  
class ostream_iterator {  
    ostream & os;  
    string delim;  
public:  
    ostream_iterator(ostream& oos, const string& d): os(oos), delim(d){}  
    void operator++(){}  
    void operator=(const T & t) {  
        os << t << delim;  
    }  
};
```