

# Лекция 3

Операция объединения

*merge.*

Категории итераторов

# Алгоритм *merge* (объединение)

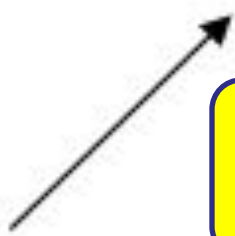
Рассмотрим операцию объединения *a* и *b* в *c* :

Контейнер *a*  
2 3 8 20 25



Результирующий контейнер *c*  
2 3 7 8 9 20 23 25 28 30 33

Контейнер *b*  
7 9 23 28 30 33



Каждая из объединяемых последовательностей упорядочена!!!

Алгоритм *merge* можно использовать для каждого из 4 типов последовательных контейнеров (массивы, векторы, двусторонние очереди и списки). Три участника алгоритма (*a*, *b* и *c* на рисунке ) не обязаны принадлежать к одному и тому же контейнерному типу.

# Объединение вектора и массива в список

```
#include <iostream>
#include <vector>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;
int merge()
{ vector<int> a(5); a[0] = 2; a[1] = 3; a[2] = 8;
a[3] = 20; a[4] = 25;
int b[6] = {7, 9, 23, 28, 30, 33};
list<int> c; // Список сначала пуст
merge(a.begin(), a.end(), b, b+6, inserter(c, c.begin()));
list<int>::iterator i;
for (i=c.begin(); i != c.end(); ++i)
cout << *i << " ";
cout << endl; return 0; }
```

## Замечания

Здесь также нужно использовать итератор вставки, если хотим писать в **c** в режиме вставки. В качестве альтернативы мы могли бы написать:

```
list<int> c (11); // принимаем 5 + 6 = 11 элементов  
merge(a.begin(), a.end(), b, b+6, c.begin());
```

выделив достаточно места при определении принимающего списка **c**.

Сам по себе алгоритм **merge** работает *в режиме замещения*, то есть не создает новых элементов контейнера, а помещает значения в существующие. Чтобы вставлять новые элементы (*режим вставки*) при объединении, мы должны использовать *вставляющий итератор*, как показано в полной программе. В любом случае результат работы программы следующий:

2 3 7 8 9 20 23 25 28 30 33

## Типы, определенные пользователем

Кроме стандартных типов, таких как *int*, в контейнерах STL можно хранить типы, определенные пользователем.

Так как вызов *merge(...)* в программе *merge.cpp* основан на операции сравнения “меньше чем”  $<$ , такой вызов для новых типов возможен, только если мы для этих типов определяем *operator<*.

Рассмотрим это на простом примере.

## Типы, определенные пользователем

```
// merge2.cpp: Объединяем записи, используя
// имена в качестве ключей.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
// Описание operator< находится внутри
// структуры entry
struct entry
{ long nr; char name[30];
  bool operator<(const entry &b)const
  { return strcmp(name, b.name) < 0; }
};
```

## В качестве ключей используются имена

```
int merge2()
{
    entry a[3] = {{10, "Betty"},
                  {11, "James"},
                  {80, "Jim"}},
    b[2] = {{16, "Fred"},
            {20, "William"}},
    c[5], *p;
    merge(a, a+3, b, b+2, c);
    for (p=c; p != c+5; p++)
        cout << p->nr << " " << p->name << endl;
    cout << endl; return 0;
}
```

```
10 Betty
16 Fred
11 James
80 Jim
20 William
```

Для работы программы существенно, что имена в каждом из массивов **a** и **b** располагаются в алфавитном порядке. Программа объединяет **a** и **b** в **c** (в алфавитным порядком имен)

## В качестве ключей используются числа

Чтобы числа шли в порядке возрастания, их нужно было бы перечислить в таком порядке в заданных массивах *a* и *b* и, кроме того, заменить определение *оператора “меньше”*. Поскольку числа и так уже расположены в порядке возрастания в обоих массивах, нам остается только вместо функции-члена *operator <* использовать следующую:

**struct entry**

```
{ long nr; char name[30];  
bool operator<(const entry &b) const  
    {return nr < b.nr;  
    }  
};
```



## В качестве ключей используются числа

Результат:

10 Betty  
11 James  
16 Fred  
20 William  
80 Jim

Функция ***operator<*** не обязана быть членом класса (или структуры) ***entry***. Иными словами, мы могли бы написать:

```
struct entry
```

```
{
```

```
    long nr;
```

```
    char name[30];
```

```
};
```

```
bool operator<(const entry &a, const entry &b) const
```

```
{    return strcmp(a.name, b.name) < 0;
```

```
}
```

## Категории итераторов

Мы можем использовать алгоритм *sort* (произвольный доступ) для массивов, векторов и двусторонних очередей, но не для списков. Алгоритм *find*, напротив, может быть использован для всех четырех типов последовательных контейнеров.

В обоих случаях используются итераторы, но алгоритм *sort* требует “более мощных” итераторов, нежели алгоритм *find*. Итераторы можно поделить на пять категорий, в соответствии с теми операциями, которые для них определены.

Предположим, что *i* и *j* - итераторы одного вида.

Тогда, основные операции, выполняемые с любым итератором:

□ Разыменованное итератора; если *i* - итератор, то *\*i* – значение объекта, на который он ссылается.

□ Присваивание одного итератора другому: *i = j*.

□ Сравнение итераторов: *i == j*, *i != j*.

□ Перемещение итератора по всем элементам контейнера: с помощью префиксного (*++i*) или постфиксного (*i++*) инкремента.

Т.к. реализация итератора специфична для каждого класса, то при объявлении итераторов указывается область видимости:

```
vector<int>:: iterator iv;
```

Пусть *i* – итератор; просмотр элементов контейнера записывается так:

```
for (i = first; i != last; ++i)
```

где *first* – значение итератора, указывающего на первый элемент контейнера, *last* – значение итератора, указывающего на воображаемый элемент, следующий за последним элементом контейнера. Операция сравнения **<** заменена на операцию **!=**, поскольку операции **<** и **>** для итераторов в общем случае не поддерживаются.

Методы *begin()* и *end()* возвращают адреса *first* и *last* соответственно.

# Таблица операций, применимых к итераторам

**x** - переменная того же типа, что и элементы рассматриваемого контейнера, а **n** – *int*.

Категория итератора	<u>Операции</u> (дополнительно к $\underline{i} == j, \underline{i} != j, \underline{i} = j$ )	Какие контейнеры предоставляют	Каким алгоритмом используется
входной (input)	$x = * \underline{i}, ++ \underline{i}, \underline{i}++$	все четыре	<i>find</i>
выходной (output)	$* \underline{i} = x, ++ \underline{i}, \underline{i}++$	все четыре	<i>copy</i> (приемник)
прямой (forward)	как у входного и выходного сразу	все четыре	<i>replace</i>
двунаправленный (bidirectional)	как у прямого и $-- \underline{i}, \underline{i}--$	все четыре	<i>reverse</i>
произвольного доступа (random access)	как у двунаправленного и $\underline{i} + n, \underline{i} - n, \underline{i} += n, \underline{i} -= n, \underline{i} < j, \underline{i} > j, \underline{i} <= j, \underline{i} >= j$	массив, <i>vector</i> , <u><i>deque</i></u> (но не <i>list</i> )	<u><i>sort</i></u>

## Категории итераторов

- **Входные (*input*)** итераторы используются для чтения значений из контейнера, аналогично вводу данных из потока *cin*.
- **Выходные (*output*)** итераторы используются для записи значений в контейнер, аналогично выводу данных в поток *cout*.
- **Прямые (*forward*)** итераторы поддерживают навигацию по контейнеру только в прямом направлении, позволяют читать и изменять данные в контейнере.
- **Двунаправленные (*bidirectional*)** итераторы в дополнение ко всем операциям прямых итераторов, поддерживают навигацию и в прямом, и в обратном направлениях (реализованы операции префиксного и постфиксного уменьшения).

## Категории итераторов

□ Итераторы *произвольного доступа* (*random access*) получили, добавив к двунаправленному итератору операции  $+$ ,  $-$ ,  $+=$ ,  $-=$ ,  $<$ ,  $>$ ,  $<=$ ,  $>=$ , т.е. доступа к произвольному элементу контейнера. Добавление целого числа к итератору возможно только для итераторов произвольного доступа; эта операция требуется, к примеру, для алгоритма сортировки.

Так как *список* не предоставляет итераторов произвольного доступа, мы не можем применить к списку алгоритм *sort*.

Теперь понятно, что выражение  $i - 1$  допустимо *только для операторов произвольного доступа*, тогда как  $--i$  поддерживается также двунаправленными итераторами.

## Демонстрация итераторов произвольного доступа:

```
int a[3] = {5, 8, 2};  vector<int> v(a, a+3);
```

```
vector<int>::iterator iv = v.begin(), ivl;
```

```
ivl = iv + 1;
```

```
bool b1 = iv < ivl;
```

```
// + и < допустимы, поскольку
```

```
// iv и ivl – итераторы произвольного доступа.
```

## Демонстрация двунаправленных итераторов:

```
list<int> w(a, a+3);
```

```
list<int>::iterator iw = w.begin(), iwl;
```

```
iwl = iw + 1;          // Ошибка
```

```
bool b2 = iw < iwl;   // Ошибка
```

```
// + и < недопустимы, поскольку
```

```
// iw и iwl – двунаправленные итераторы.
```

```
// Следующие две строчки являются правильными:
```

```
iwl = iw;
```

```
bool b3 = iw == iwl; // b3 =true
```



## Категории итераторов и алгоритмы

Алгоритм *find* требует исключительно тех операций над итераторами, которые определены для *входных итераторов*, потому что ему достаточно только читать элементы последовательности, исполняя, например, операцию присваивания  $x = *i$ .

Алгоритм *replace*, заменяющий одно определенное значение на другое требует только *прямых итераторов* с соответствующими операциями.

*На двунаправленных итераторах* базируются алгоритмы, выполняющие реверсивные операции, например, алгоритм *reverse*. Этот алгоритм меняет местами все объекты в цепочке, на которую ссылаются переданные ему итераторы.

## Категории итераторов и алгоритмы

Итераторы **двунаправленного доступа** возвращаются несколькими контейнерами STL: ***list, set, multiset, map*** и ***multimap***.

**Итераторы произвольного доступа** возвращают такие контейнеры, как ***vector*** и ***deque***.

Итераторы произвольного доступа – самые "умелые" из основных итераторов и, как правило, все сложные алгоритмы, требующие расширенных вычислений, оперируют с этими итераторами.

Создаем переменную типа **"*итератор произвольного доступа*"**. Для этого берем итератор и на его основе создаем другой, более удобный:

```
typedef vector<int>::iterator vectltr;  
vectltr itr ;
```

# Потоковые итераторы

Можно применить алгоритм *copy* для вывода:

```
#include <iterator>
```

```
const int N = 4;   int a[N] = {7, 6, 9, 2};
```

```
copy(a, a+N, ostream_iterator<int>(cout, " "));
```

Это можно сделать таким образом:

```
ostream_iterator<int> i (cout, " " );
```

```
copy (a, a+N, i) ; // потоковый итератор – 3-й аргумент
```

В поток стандартного вывода *cout* будут выведены числа 7, 6, 9 и 2, как если бы написали:

```
for (int* p=a; p!= a+N; p++)  
    cout << *p << " ";
```

Для ввода используем аналогичный прием:

```
istream_iterator<int> is(cin);
```

## Потоковые итераторы

Программа читает из стандартного потока *cin* числа, вводимые пользователем и дублирует их на экране. Работа программы заканчивается, при вводе числа 77:

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
void prog1()
{ istream_iterator<int> is(cin);
  ostream_iterator<int> os(cout, " ");
  int input;
  while((input = *is) != 77)
  { *os++ = input;
    is++ ;
  }
}
```

## Операции с итераторами

К итераторам произвольного доступа можно применять арифметические операции:

```
int n, dist; // i и i0 – итераторы произвольного доступа
i0 = i; i += n; dist = i - i0; // dist == n
```

Существуют функции ***advance*** и ***distance***:

```
int n, dist; // i и i0 – итераторы, но необязательно
// произвольного доступа
```

```
i0 = i; advance(i, n); dist = 0;
distance(i0, i, dist); // dist == n
```

***advance(i, n)***       $\rightarrow i += n$

***distance(i0, i, dist)***       $\rightarrow dist = i - i0$

Функция ***distance*** служит для определения расстояния между элементами контейнера.

Операции ***advance*** и ***distance*** будут выполняться гораздо быстрее для итераторов произвольного доступа, чем для итераторов других типов.

## Алгоритм *replace*

*replace* позволяет найти все элементы с определенным значением в заданном контейнере и заменить их другим значением.

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
int replace_massiv()
{ char str[ ] = "abcabcabc";
int n = strlen(str);
replace(str, str+n, 'b', 'q');
cout << str << endl;
return 0;
}
```

Заменяет все элементы 'b' на 'q', результат:  
**aqcaqcaqc**

## Алгоритм *reverse*

*reverse* позволяет заменить последовательность на обратную ей. Этот алгоритм требует двунаправленных итераторов, которые предоставляют все четыре контейнера.

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
int reverse_massiv()
{ char str[ ] = "abcklmxyz";
reverse(str, str+strlen(str));
cout << str << endl;
// Будет выведено: zyxmlkcsba
return 0;
}
```

# Краткие итоги

Рассмотрели алгоритмы:

1. **merge()** – объединение элементов различных контейнеров.

**merge(a.begin(), a.end(), b, b+6, inserter(c, c.begin()));** – в режиме вставки.

**merge(a.begin(), a.end(), b, b+6, c.begin());** – в режиме замещения.

Каждая из объединяемых последовательностей упорядочена!!! Иначе алгоритм не будет правильно работать.

**operator<** – оператор “меньше”, определяет операции упорядочивания (сравнения) для типов, определенных пользователем.

**merge(a, a+3, b, b+2, c);** – рассматривали структуры, объединение производилось по одному из ключей (либо по имени, либо по номеру). Именно оператор **operator<** задаёт для алгоритма **merge()** вид упорядочивания.



## Краткие итоги

2. ***replace()*** – замена одних элементов на другие  
***replace (str, str+n, 'b', 'q');***

3. ***reverse*** позволяет заменить последовательность на обратную ей.

***reverse (str, str+strlen(str));***

## Категории итераторов

▣ ***Входные (input)*** итераторы используются для чтения значений из контейнера. ***find()***

▣ ***Выходные (output)*** итераторы используются для записи значений в контейнер. ***copy()***

▣ ***Прямые (forward)*** итераторы поддерживают навигацию по контейнеру только в прямом направлении, позволяют читать и изменять данные в контейнере.

***replace()***

# Краткие итоги

## Категории итераторов

□ **Двунаправленные (*bidirectional*)** итераторы в дополнение ко всем операциям прямых итераторов, поддерживают навигацию и в прямом, и в обратном направлениях. *reverse()*

□ **Итераторы произвольного доступа (*random access*)** получили, добавив к двунаправленному итератору операции +, -, +=, -=, <, >, <=, >=, т.е. доступа к произвольному элементу контейнера. *sort()*