

# **Тема 7**


# **Алгоритмы**

# **библиотеки STL**

# Вспомогательные понятия

**Последовательность** – набор однотипных элементов данных, для которых определено понятие «следующий элемент последовательности» (это понятие определено для всех элементов, кроме последнего).

Примеры последовательностей:

- ▶ массивы
  - ▶ контейнерные классы из библиотеки STL
  - ▶ строки (класс `string`)
  - ▶ потоки
- 

# Вспомогательные понятия

**Итератор** – инструмент для доступа к элементам последовательности

**Указатель (имя массива)** – частный случай итератора, если в качестве последовательности используется массив

**Интервал** – идущие подряд элементы последовательности. Интервал задается парой итераторов. Первый итератор связан с первым элементом интервала, второй итератор – с первым элементом, следующим за интервалом. Второй итератор может быть недействительным

# Примеры

## Сортировка массива

```
long M[50];  
// заполнение массива  
sort(M, M+50);
```

## Сортировка контейнера vector

```
vector <long> M;  
// заполнение контейнера  
sort(M.begin(), M.end());
```

## Подсчет количества элементов

```
long *p = new long [20];  
deque <long> d;  
cout << count(p, p+20, 5) << endl;  
cout << count(d.begin(), d.end(), 5) << endl;
```

# Виды итераторов

Тип итератора	Доступ	Разыменован- вание	Итерация	Сравнение
Итератор вывода (output iterator)	Только запись	*	++	
Итератор ввода (input iterator)	Только чтение	*, ->	++	==, !=
Прямой итератор (forward iterator)	Чтение и запись	*, ->	++	==, !=
Двунаправленный итератор (bidirectional iterator)	Чтение и запись	*, ->	++, --	==, !=
Итератор с произвольным доступом (random-access iterator)	Чтение и запись	*, ->, []	++, --, +, -, +=, -=	==, !=, <, <=, >, >=


# Типы итераторов, поддерживаемые контейнерами

Тип итератора	vector	list	deque	set	multiset	map	multimap
Произвольного доступа	X		X				
Двунаправленный	X	X	X	X	X	X	X
Прямой	X	X	X	X	X	X	X
Входной	X	X	X	X	X	X	X
Выходной	X	X	X	X	X	X	X

# Типы итераторов, требующиеся представленным алгоритмам

Алгоритм	Входной	Выходной	Прямой	Двунаправленный	Произвольного доступа
for_each	x				
find	x				
count	x				
copy	x	x			
replace			x		
unique			x		
reverse				x	
sort					x
nth_element					x
merge	x	x			
accumulate	x				

# Случаи возникновения недействительных итераторов

- ▶ итератор не был инициализирован;
  - ▶ итератор указывает на конец последовательности;
  - ▶ элемент контейнера, с которым он связан, удален;
  - ▶ контейнер, с которым он связан, изменил размеры или уничтожен.
- 



# Функциональные объекты

**Функциональный объект** - класс, в котором определена операция вызова функции.

**Синонимы:** объект-функция, функтор

**Предикат** – функциональный объект (или функция), возвращающий логическое значение.

**Пример ФО:**

```
class isbest{
public:
    int operator() (int a, int b){
        return (a>b || a%10==0 ? a : b);
    }
};
```

```
...
isbest b;
int x, y;
cout << b(x, y);
```

# Часто используемые виды функциональных объектов

бинарная функция	$T_3 (T_1, T_2)$
унарная функция	$T_3 (T_1)$
бинарный предикат	$\text{bool} (T_1, T_2)$
унарный предикат	$\text{bool} (T_1)$

Здесь  $T_1$ ,  $T_2$ ,  $T_3$  – произвольные типы

# Использование функциональных объектов в алгоритмах

## Задача:

подсчитать количество элементов вектора, описанного как

```
vector <long> d;
```

больших 5

Алгоритм **count\_if** определяет число элементов интервала, для которых верен заданный унарный предикат (функция либо функциональный объект)

```
int count_if (интервал, унарный_предикат)
```

# Примеры использования функциональных объектов

## Первый вариант решения (функция)

```
bool gr5(long a) { return (a>5); }
```

...

```
cout << count_if (d.begin(), d.end(), gr5) << endl;
```

## Второй вариант решения (функциональный объект)

```
class _gr5 {
```

```
public:
```

```
    bool operator() (long a) { return (a>5); }
```

```
};
```

...

```
cout << count_if (d.begin(), d.end(), _gr5())  
    << endl;
```

# Примеры использования функциональных объектов

Третий вариант решения (с использованием шаблона функции)

```
template <long value>
bool gr5(long a)
{
    return (a > value);
}
...
cout << count_if(d.begin(), d.end(), gr5<10>)
      << endl;
```

# Примеры использования функциональных объектов

Четвертый вариант решения  
(функциональный объект, выражение вместо константы)

```
class _gr {  
    long _n;  
public:  
    _gr(long an) { _n = an; } //конструктор  
    bool operator() (long a) { return (a>_n); }  
};  
  
...  
    long val = 10;  
...  
    cout << count_if (d.begin(), d.end(), _gr(val))  
        << endl;
```

# Шаблоны стандартных функциональных объектов

Описаны в заголовочном файле <functional>

## Арифметические функциональные объекты

Название	Вид	Результат
plus	$T(T, T)$	$x + y$
minus	$T(T, T)$	$x - y$
multiplies	$T(T, T)$	$x * y$
divides	$T(T, T)$	$x / y$
modulus	$T(T, T)$	$x \% y$
negate	$T(T)$	$-x$

# Шаблоны стандартных функциональных объектов

## Предикаты стандартной библиотеки

Название	Вид	Результат
equal_to	bool (T, T)	$x == y$
not_equal_to	bool (T, T)	$x != y$
greater	bool (T, T)	$x > y$
less	bool (T, T)	$x < y$
greater_equal	bool (T, T)	$x \geq y$
less_equal	bool (T, T)	$x \leq y$
logical_and	bool (T, T)	$x \&\& y$
logical_or	bool (T, T)	$x    y$
logical_not	bool (T)	$! x$



# Примеры использования стандартных функциональных объектов

## 1) Критерий сортировки

Для ассоциативных контейнеров критерий – по возрастанию

`less<>` (меньше), следующие записи эквивалентны:

```
set <int> coll;
```

```
set <int, less<int> > coll;
```

По убыванию `greater<>`:

```
set <int, greater<int> > coll;
```

# Примеры использования стандартных функциональных объектов

## 2) Изменение знака на противоположный

```
template <class X>
class PrintElement
{
    public:
        void operator() (X elem) const
        {
            cout << elem << ' ';
        }
};
```

# Примеры использования стандартных функциональных объектов

## Продолжение

```
void main()
{
    list<int> coll;
    // Заполняем элементы коллекции
    for (int i = 1; i < 5; i++)
    {
        coll.push_back(i);
        coll.push_front(-i);
    }
    for_each(coll.begin(), coll.end(),
             PrintElement<int>());
    cout << "\n";
}
```

# Примеры использования стандартных функциональных объектов

// Изменяем знак элементов коллекции

```
transform(coll.begin(), coll.end(),  
          coll.begin(),  
          negate<int>());
```

// Выводим элементы коллекции

```
for_each(coll.begin(), coll.end(),  
         PrintElement<int>());  
cout << "\n";
```

```
}
```

Ответ

```
-4 -3 -2 -1 1 2 3 4  
4 3 2 1 -1 -2 -3 -4
```

# Отрицатели и связыватели

**Адаптером функции** называют функцию, которая получает в качестве аргумента функцию и конструирует из нее другую функцию. На месте функции может быть также функциональный объект.

Стандартная библиотека содержит описание нескольких типов адаптеров:

- **связыватели** для использования функционального объекта с двумя аргументами как объекта с одним аргументом;
- **отрицатели** для инверсии значения предиката.

# Отрицатели и связыватели

Название	Вид	Результат
not1	<code>bool (bool(T))</code>	отрицание унарного предиката
not2	<code>bool (bool (T, T))</code>	отрицание бинарного предиката
bind2nd	<code>bool(bool (T t1, T t2), const T t3)</code>	преобразует бинарный предикат в унарный, подставляя значение t3 вместо t2 и t1 соответственно
bind1st	<code>bool(bool (T t1, T t2), const T t3)</code>	

## Пример использования отрицателей

Для инверсии `less<int>()` нужно записать

```
not2(less<int>())
```

Эквивалентно `greater_equal <int>()`

## Пример использования стандартных функциональных объектов и связывателей

Пятый вариант решения рассмотренной ранее задачи:

```
long k;
```

```
...
```

```
cout << count_if (d.begin(), d.end(),  
bind2nd(greater <long> (), 5)) << endl;
```

# Пример использования связывателей

Вычислить количество элементов целочисленного массива, меньших 40:

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
void main()
{
    int m[8] = {45, 65, 36, 25, 674, 2, 13, 35};
    cout << count_if(m, m + 8, bind2nd(less<int>(), 40));
}
```

Функция `bind2nd` превращает условие сравнения `x < y` в условие `x < 40`.



# Классификация алгоритмов

- ▶ немодифицирующие алгоритмы;
- ▶ модифицирующие алгоритмы;
- ▶ алгоритмы удаления;
- ▶ перестановочные алгоритмы;
- ▶ алгоритмы сортировки;
- ▶ алгоритмы упорядоченных интервалов;
- ▶ численные алгоритмы.

Некоторые алгоритмы могут принадлежать сразу нескольким категориям.

# Особенности алгоритмов

## Немодифицирующие алгоритмы

- ▶ не изменяется ни последовательность, ни ее элементы.

## Модифицирующие алгоритмы

- ▶ структура последовательности не изменяется, изменяются только ее элементы;
- ▶ после «удаления» некоторых элементов освободившееся место заполняется мусором.

## Общее

- ▶ передается интервал последовательности, заданный парой итераторов;
- ▶ контроля за выходом за границы последовательности нет;
- ▶ алгоритмы работают в режиме замены, а не в режиме вставки. Специальные итераторы вставки `inserter` и `back_inserter` переводят алгоритм в режим вставки.

# Особенности алгоритмов

Для повышения мощности и гибкости некоторые алгоритмы позволяют передавать пользовательские объект-функции или предикаты:

- ▶ унарный предикат может передаваться алгоритму поиска в качестве критерия поиска. Унарный предикат проверяет, соответствует ли элемент заданному критерию;
- ▶ бинарный предикат может передаваться алгоритму сортировки в качестве критерия сортировки. Бинарный предикат сравнивает два элемента;
- ▶ унарный предикат может использоваться как критерий, определяющий, к каким элементам должна применяться операция;
- ▶ предикаты используются для модификации операций в численных алгоритмах.

# Назначение алгоритмов

- ▶ суффикс `_if` используется при наличии двух похожих форм алгоритма с одинаковым количеством параметров; первой форме передается значение, а второй — функция или объект функции. В этом случае версия без суффикса `_if` используется при передаче значения, а версия с суффиксом `_if` — при передаче функции или объекта функции.
- ▶ суффикс `_copy` означает, что алгоритм не только обрабатывает элементы, но и копирует их в приемный интервал. Например, алгоритм `reverse()` переставляет элементы интервала в обратном порядке, а `reverse_copy()` копирует элементы в другой интервал в обратном порядке.

**Спасибо за внимание**