Алгоритмы STL

- Алгоритмы выполняют типовые операции над элементами контейнеров.
- Алгоритмы отделены от самих контейнеров, чтобы можно было использовать одни и те же алгоритмы с разными контейнерами
- Алгоритмы реализованы в виде шаблонных функций в файле algorithm
- Функции, как правило, принимают в качестве параметров интервал, заданный итераторами, и параметры, задающие режим работы.
- Алгоритмы можно использовать и с обычными массивами; «конец» интервала определяется как следующий за последним элемент.

```
template<typename ITERATOR, typename RESULT>
RESULT an_algorithm(ITERATOR first, ITERATOR last, ...);
```

<algorithm>

Типовая задача	**Примеры функций**
Поиск	find, find_if, search, search_n, find_end, min_element, max_element
Преобразование последовательности	transform, replace, generate, rotate, reverse, ..
Обход элементов	for_each
Обмен значениями, копирование	swap, iter_swap, copy, ..
Упорядочивание элементов	sort, partition, unique, partial_sort, ..
Работа с упорядоченными интервалами	binary_search, merge, includes, lover_bound, upper_bound, equal_range, set_union, set_intersection, set_difference,..
Агрегирование (редукция)	cout, accumulate, inner_product, ..
Проверка условия для последовательности	all_of, any_of, none_of, equal, ..

Алгоритмы поиска

- Алгоритмы требуют итераторов последовательности

```
list<int> scores { 11, 5, 9, 4, 5, 2, 7, 10, 4 };
// получаем итератор, найденного минимума
auto it = min_element(scores.begin(), scores.end());
cout << "min element: " << *it << "\n";
cout << "position: " << distance(scores.begin(), it)  << "\n";
```

- Функции поиска в случае «неудачи» возвращают *конечный итератор*

```
auto found = find(scores.begin(), scores.end(), 12);
if(found == scores.end())
        cout << "number was not found\n";
```

- Вместо итераторов можно использовать указатели для работы с массивом

```
int numbers[] {1, 3, 4, 2, 6, 6, 7, 9, 5, 11};
int* pos = find(numbers, numbers + 10, 6);
cout << "found at " << (pos - numbers) << " position\n";
```

Настройка алгоритмов

- Как правило, функции-алгоритмы имеют несколько перегрузок: «простые» версии для встроенных типов и «сложные» версии для пользовательских типов, требующие *дополнительной информации* о пользовательском типе или о *специфике* обработки

```
struct wh { float weight; float height; };
list<wh> patients;
auto it2 = max_element(begin(patients), end(patients), .. );
// ищем элемент, удовлетворяющий условию
int* p find_if(numbers, numbers + 10, ..);
```

- Дополнительная информация:
  - как сравнивать объекты на «больше»-«меньше» (sort, min_element);
  - как сравнивать объекты на равенство (equal);
  - условие отбора элементов (find_if);
  - что делать с каждым элементом (for_each, transform, accumulate);
- Такая информация передается в алгоритм в виде *функционального объекта*

Функциональные объекты

- Функциональный объект – это такая сущность, которая *ведет себя как функция* (= можно «вызывать») и как переменная (= можно передавать в качестве аргумента).
- Основное назначение функциональных объектов – это возможность *настройки других функций* (= управление поведением).
- Возможность работы с функциями как с обычными переменными является одной из ключевых особенностей **функционального программирования**
- В качестве функциональных объектов в C++ могут использоваться:
  - функции;
  - экземпляры классов, имеющих operator ();
  - предопределенные функциональные объекты в STL;
  - лямбда-выражения;

Функции

- В Си можно оперировать функциями как обычными переменными (передавать в качестве аргумента, создавать массив функций);

```
// обычная функция
bool isGreaterThan7(int item) { return item > 7; }
void withFuncObjects
{
    int numbers[] {1, 3, 4, 2, 6, 6, 7, 9, 5, 11};
    // ищем элемент, удовлетворяющий условию
    int* p = find_if(numbers, numbers + 10, isGreaterThan7);
    cout << "Item " << *p << " at position " << p - numbers << endl;
}
```

- Имя функции интерпретируется как адрес; его можно сохранить в переменной – указателе на функцию

```
// f - указатель на функцию
bool (*f)(int) = isGreaterThan7;
f(4);
```

Функциональный объект как экземпляр класса

- В качестве функционального объекта можно использовать класс с оператором ()
- В таком объекте можно ввести поля, которые инициализируются при использовании объекта

```
class GreaterThan
{
    int param;
    public:
        GreaterThan(int par): param(par) {}
        bool operator() (int n)
        {
            return n > param;
        }
};
```

```
// используем функциональный объект
p = find_if(numbers, numbers + 10, GreaterThan { 6 });
// подсчёт количества элементов, удовлетворяющих условию
int res = count_if(numbers, numbers + 10, GreaterThan {5} );
```

Предопределённые функциональные объекты <functional>

Функциональный объект		**Операция**
Арифметические операции	negate<T>	- x
	plus<T>	x + y
	minus<T>	x – y
	multiplies<T>	x * y
	divides<T>	x / y
	modulus<T>	x % y
Сравнение	equal_to<T>	x == y
	not_equal_to<T>	x != y
	less<T>, less_equal<T>	x < y, x <= y
	greater<T>, greater_equal<T>	x > y, x >= y
Логические операции	logical_and<T>, logical_or<T>,	x && y, x || y,

```
template<typename TYPE>
struct plus {
    TYPE operator() (const TYPE& p1, const TYPE& p2) const{ return p1 + p2; }
};
```

transform + встроенные «предикаты»

```
vector<int> v1 { 3, 4, -6, 2, 1, 9, -34, -3};
vector<int> v2(v1.size());
// преобразование элементов с помощью унарного предиката
transform(v1.begin(), v1.end(), v2.begin(), negate<int>());

// поэлементное комбинирование с предикатом
// [beg1, end1) для первой последовательности
// beg2 - для второй, beg3 - для результата
// BinaryPredicate - как соединять элементы
transform(v1.begin(), v1.end(), v2.begin(), v2.begin(), plus<int>());
```

Агрегирование: N -> 1

```
vector<int> v { 1, 2, 3, 4, 5};
int sum, prod;
// по умолчанию вычисляется сумма result = 0 + v[0] + v[1] + ..
sum = accumulate(v.begin(), v.end(), 0);
// result = 1 * v[0] * v[1] * ..
prod = accumulate(v.begin(), v.end(), 1, multiplies<int>());
// работаем со строками line = “” + “old ” + “McDonald ”
vector<string> words { "old ", "McDonald ", "had ", "a ", "farm "};
string line = accumulate(words.begin(), words.end(), string {}, plus<string>());
```

агрегирование: N x N -> 1

```
// res = initValue + Summ(v1[i] * v2[i])
int res = inner_product(v1.begin(), v1.end(), v2.begin(), 0);

vector<float> y1 { 50.6, 52.3, 70.7, 43.9, 40.2, 68.0};
vector<float> y2 { 51.3, 54.5, 73.1, 42.8, 43.4, 70.8};
// res = res op1 (y1 op y2)
float error = inner_product(y1.begin(), y1.end(),
                            y2.begin(),
                            0, // Начальное значение
                            plus<float>(), // как агрегируем результаты
                            deviation); // как обрабатываем элементы
```

```
float deviation(float y1, float y2)
{
    float d = y1 - y2;
    return d * d;
}
```

Адаптеры для встроенных предикатов

- С помощью *функциональных адаптеров* можно конвертировать бинарные операции в унарные

```
        // используем встроенный функциональный объект с адаптером
        p = find_if(numbers, numbers + 10,  bind2nd(less_equal<int>(), 5));

vector<string> animals { "cow", "horse", "pig", "duck", "chicken"};
vector<string> a(animals.size());
// к каждому элементу добавляем «s» («horse» -> «horses»)
transform(animals.begin(), animals.end(), a.begin(),
                                    bind2nd(plus<string>(), string {"s"}));

// ко всем, кроме последнего, добавляем «, »
transform(a.begin(), a.end()-1, a.begin(),
                                    bind2nd(plus<string>(), string {", "}));

string farm = accumulate(a.begin(), a.end(), string {}, plus<string>());
cout << "and on that farm he had: " << farm << endl;
```

Лямбда-выражения в C++

- Лямбда-выражения представляют собой *функциональные сущности*, которые определяются внутри другой функции
- Лямбда-выражения используют, когда код функции не обладает самостоятельной ценностью, а нужен только в месте использования
- Лямбда-выражения часто позволяют сделать код более лаконичным и удобным для восприятия

```
vector<int> v(10);
// используем библиотечную функцию для генерации случайных чисел
generate(v.begin(), v.end(), rand);

// обрабатываем каждый элемент с помощью анонимной функции
for_each(v.begin(), v.end(), [](int n) { cout << n << endl;});
```

Лямбда-выражения в C++

- Лямбда-выражения – краткая форма записи анонимных локальных функциональных объектов

```
// обрабатываем каждый элемент с помощью анонимной функции
for_each(v.begin(), v.end(), [](int n) { cout << n << endl;});
```

```
class MyLambda
{
public:
    void operator() (int n) const { cout << n << endl; }
};
```

Формат лямбда-выражения в C++

[захват_внешних_переменных **] (** аргументы_функции **) ->** Тип_возврата

{

Тело_анонимной_функции

}

- Не все блоки обязательно заполнять

```
// empty-lambda
[](){};
```

- Лямбда-выражения, как правило, используются как аргумент функции. Сигнатура этой функции имеет определенные «ожидания» от лямбда-выражения: что может быть результатом и какие аргументы требуются
- В таких случаях компилятор автоматически определяет тип возвращаемого значения, число и типы аргументов

Возвращаемое значения для лямбда-выражения

- Автоматический вывод типа для возвращаемого значения

```
vector<string> w { "apple", "lemon", "cucumber", "onion" };
// контекст требует от лямбды сигнатуры  void f (string)
for_each(w.begin(), w.end(), [](string word) { cout << word << endl; });



// автоматический вывод типа возврата по return
transform(w.begin(), w.end(), lengths.begin(), [](string word)
{
   return word.length();
});
```