

Алгоритмы и функциональные объекты

Алгоритмы и функциональные объекты	1
Перегрузка операции вызова функции	2
Функциональные объекты	4
Пользовательские функции вместо функциональных объектов	8
Не модифицирующие операции с последовательностями	13
Алгоритм adjacent_find	14
Алгоритм find()	14
Заголовочные файлы	16
Диапазоны	16
Алгоритм count()	17
Алгоритм sort()	18
Алгоритм search()	18
Алгоритм merge()	20
Добавление _if к аргументам	22
Алгоритм for_each ()	24
Алгоритм transform()	25

Алгоритмы STL выполняют различные операции над наборами данных. Они были разработаны специально для контейнеров, но их замечательным свойством является то, что они применимы и к обычным массивам C++. Это может сильно упростить работу с массивами. К тому же изучение алгоритмов для контейнеров поможет усвоить общие принципы подобной обработки данных, безотносительно к контейнерам и STL.

Алгоритмы STL предназначены для работы с контейнерами и другими последовательностями. Каждый алгоритм реализован в виде шаблона или набора шаблонов функции, поэтому может работать с различными видами последовательностей и данными разнообразных типов. Для настройки алгоритма на конкретные требования пользователя применяются функциональные объекты.

Перегрузка операции вызова функции

Класс, в котором определена операция вызова функции, называется функциональным. От такого класса не требуется наличия других полей и методов:

```
class if_greater
{
    public:
        int operator() (int a, int b) const { return a > b;}
};
```

Использование такого класса имеет весьма специфический синтаксис. Рассмотрим пример:

```
if_greater x;
cout « x(1, 5) « endl;      // Результат - 0
cout « if_greater()(5, 1) « endl;    // Результат - 1
```

Поскольку в классе if_greater определена операция вызова функции с двумя параметрами, выражение x(1, 5) является допустимым (то же самое можно записать в виде x.operator () (1, 5). Как видно из примера, объект функционального класса используется так, как если бы он был функцией.

Во втором операторе вывода выражение if_greater() используется для вызова конструктора по умолчанию класса if_greater. Результатом выполнения этого выражения является объект класса if_greater. Далее, как и в предыдущем случае, для этого объекта вызывается функция с двумя аргументами, записанными в круглых скобках.

Операцию () (а также операцию []) можно определять только как метод класса. Можно определить перегруженные операции вызова функции с различным количеством аргументов. Функциональные объекты широко применяются в стандартной библиотеке C++. Ниже приведён пример параметризованного класса Greater (шаблон классов).

```
//-----
#include "stdafx.h"
```

```

#include <iostream>
//-----

using namespace std;
//-----
template<class T>
class Greater{
public:
    bool operator()(T a,T b){return a > b;};
};
//-----

int _tmain(int argc, _TCHAR* argv[])
{
    Greater<int> x;
    cout << x(4,5) << endl; //0
    cout << x.operator()(3,2) << endl; //1
    cout << Greater<int>()(3,2) << endl; //1
    cin.get();
    return 0;
}
//-----

```

Результатом работы будет:

0

1

1

Пример функционального объекта на структуре:

```

#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

template <class T = int>
struct MyFunc
{
public:
    T operator()(const T& x, const T& y) const
    {
        return x + y * y;
    }
};

int main()
{
    cout << MyFunc<int>()(8, 5) << endl;
    return 0;
}

```

}

Функциональные объекты

Функциональным объектом называется класс, в котором определена операция вызова функции. Чаще всего эти объекты используются в качестве параметров стандартных алгоритмов для задания пользовательских критериев сравнения объектов или способов их обработки. В тех алгоритмах, где в качестве параметра можно использовать функциональный объект, можно использовать и указатель на функцию. При этом применение функционального объекта может оказаться более эффективным, поскольку операцию () можно определить как встроенную.

Стандартная библиотека предоставляет множество функциональных объектов, необходимых для ее эффективного использования и расширения. Они описаны в заголовочном файле `<functional>`. Среди этих объектов можно выделить объекты, возвращающие значения типа `bool`. Такие объекты называются предикатами. Предикатом называется также и обычная функция, возвращающая `bool`. В качестве базовых классов, которые вводят стандартные имена для типов аргументов, в библиотеке определены шаблоны унарной и бинарной функции:

```
template <class Arg, class Result>
    struct unary_function
    {
        typedef Arg      argument_type;
        typedef Result    result_type;
    };
template <class Arg1, class Arg2, class Result>
    struct binary_function
    {
        typedef Arg1      first_argument_type;
        typedef Arg2      second_argument_type;
        typedef Result     result_type;
    };
```

Функциональные объекты библиотеки являются потомками этих базовых объектов-функций. Определение стандартных имен типов для аргументов и результата необходимо для того, чтобы функциональные объекты можно было использовать совместно с адаптерами и другими средствами библиотеки.

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;

адаптеры указателей на функцию;

адаптеры методов для использования методов в алгоритмах.

Арифметические функциональные объекты

В стандартной библиотеке определены шаблоны функциональных объектов для всех арифметических операций, определенных в языке C++.

Таблица 2. Арифметические функциональные объекты

Имя	Тип	Результат
plus	бинарный	$x + y$
minus	бинарный	$x - y$
multiplies	бинарный	$x * y$
divides	бинарный	x / y
modulus	бинарный	$x \% y$
negate	унарный	$-x$

Ниже приведен шаблон объекта plus (остальные объекты описаны аналогичным образом):

```
template <class T>
struct plus : binary_function<T, T, T>{
    T operator()(const T& x, const T& y) const{
        return x + y;
    }
}
```

Предикаты

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения и логических операций, определенных в языке C++. Они возвращают значение типа bool, то есть являются предикатами.

Таблица 3. Предикаты стандартной библиотеки

Имя	Тип	Результат
<code>equal_to</code>	бинарный	<code>x == y</code>
<code>not_equal_to</code>	бинарный	<code>x != y</code>
<code>greater</code>	бинарный	<code>x > y</code>
<code>less</code>	бинарный	<code>x < y</code>
<code>greater_equal</code>	бинарный	<code>x >= y</code>
<code>less_equal</code>	бинарный	<code>x <= y</code>
<code>logical_and</code>	бинарный	<code>x && y</code>
<code>logical_or</code>	бинарный	<code>x y</code>
<code>logical_not</code>	унарный	<code>! x</code>

Ниже приведен шаблон объекта `equal_to` (остальные объекты описаны аналогичным образом):

```
template <class T>
struct equal_to : binary_function<T, T, bool>
{
    bool operator()(const T& x, const T& y) const
    {
        return x == y;
    }
};
```

Программист может описать собственные предикаты для определения критериев сравнения объектов. Это необходимо, когда контейнер состоит из элементов пользовательского типа.

Допустим, вы хотите отсортировать дек чисел по возрастанию или по убыванию. Посмотрим, как это делается.

```
//-----
// sortemp.cpp
// сортировка массива типа double по убыванию,
// используется функциональный объект greater<>()
#include "stdafx.h"
```

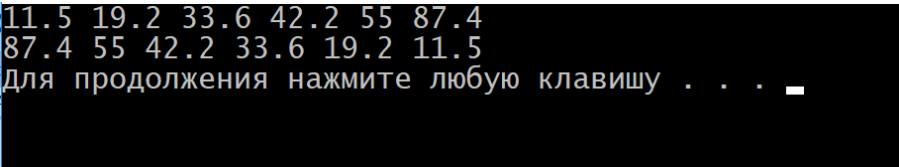
```

#include <algorithm>
#include <functional>           //для greater<>
#include <deque>
#include <iostream>
#include <windows.h>
//-----
using namespace std;
template<class T>
void Print(const T &container)
{
    for (const auto &y : container )
        cout << y << ' ';
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    double fdata[] = { 19.2, 87.4, 33.6, 55.0, 11.5, 42.2 };
    deque<double> d(fdata,fdata+6);
    //Сортировать дек по возрастанию.
    sort(d.begin(), d.end());
    // Вывести отсортированный дек.
    Print(d);
    //Сортировать дек по убыванию.
    sort( d.begin(), d.end(), greater<double>() );
    // Вывести отсортированный дек.
    Print(d);
    system("pause");
    return 0;
}
//-----

```

Алгоритм `sort()` по умолчанию сортирует по возрастанию, но использование функционального объекта `greater()` в качестве третьего аргумента `sort()` изменяет режим сортировки:



```

11.5 19.2 33.6 42.2 55 87.4
87.4 55 42.2 33.6 19.2 11.5
Для продолжения нажмите любую клавишу . . . _

```

Кроме сравнения, есть функциональные объекты для арифметических и логических операций.

В примере ниже описан функциональный объект, как шаблон функции, для использования его в алгоритме поиска (`find_if`). С помощью этого функционального объекта можно отыскивать значения, лежащие в интервале (`d..d1`).

```

//-----

```

```

#include "stdafx.h"
#include <algorithm> //для find_if
#include <fstream> //для работы с файлами
#include <iostream>
#include <vector>
using namespace std;
//-----
//функциональный объект.
template<class T = int, int d = 0, int d1 = 50>
class Diapazone{
public:
    bool operator()(T x) {return x > d&& x < d1 ;};
};
//-----

int _tmain(int argc, _TCHAR* argv[])
{
    ifstream in("inpnun.txt"); // 56 34 54 0 76 23 51 11 76 88
    vector<int> v;
    int x;
    while(in >> x, !in.eof())
        v.push_back(x);
    for(const int &y : v) cout << y << " ";
    cout << endl;
    cout << *find_if(v.begin(), v.end(), Diapazone<int, 20, 30>()) << endl; //
23
    cin.get();
    return 0;
}
//-----

```

Результатом работы программы будет значение 23.

Пользовательские функции вместо функциональных объектов

Если в вашем классе перегружен оператор ==, он и будет использоваться в алгоритме adjacent_find. В алгоритме будет вызван оператор ==, который сравнивает два объекта класса A, по умолчанию.

```

//-----
#include "stdafx.h"
#include <algorithm>
#include <iostream>
#include <sstream> //строковые потоки
#include <vector>
#include <string>
using namespace std;
//-----
class A{
    int n, d;
public:
    A(int n = 0, int d = 1): n(n), d(d){};
    string ToString()
    {
        ostringstream os;
        os << n << "/" << d;
        return os.str();
    }
}

```



```

    };
    bool operator==(const A b) const {return n*b.d == d*b.n;};
};
//-----

int _tmain(int argc, _TCHAR* argv[])
{
    vector<A> m = { { 2 }, { 3 }, { 1, 5 }, { 5 }, { 3, 4 }, { 6, 8 } };
    // Output-3/4
    cout << (*adjacent_find(m.begin(), m.end())).ToString().c_str() << endl;
    cin.get();
    return 0;
}
//-----

```

Результатом работы будет: $\frac{3}{4}$

То же самое можно сделать, если описать в классе A дружественную функцию, (см. пример ниже).

```

//-----
#include "stdafx.h"
#include <algorithm>
#include <iostream>
#include <sstream> //строковые потоки
#include <vector>
#include <string>
using namespace std;
//-----
class A{
    int n, d;
public:
    A(int n = 0, int d = 1) : n(n), d(d){};
    string ToString()
    {
        ostringstream os;
        os << n << "/" << d;
        return os.str();
    };
};

friend bool f(A& a1, A& a2) {return a1.n * a2.d == a2.n * a1.d;};
};
//-----

int _tmain(int argc, _TCHAR* argv[])
{
    vector<A> m = { { 2 }, { 3 }, { 1, 5 }, { 5 }, { 3, 4 }, { 6, 8 } };

    cout << (*adjacent_find(m.begin(), m.end(), f)).ToString() <<
endl; //c_str() Output-3/4
    cin.get();
    return 0;
}
//-----

```

Результатом работы будет: $\frac{3}{4}$

Если не один из предикатов STL вам не подходит для упорядочения значений, вы можете написать свою функцию, как это показано в примере ниже. Здесь упорядочение идёт в порядке уменьшения суммы разрядов целого числа.

```
#include "stdafx.h"
#include <algorithm>
#include <iostream>
#include <sstream> //строковые потоки
#include <deque>
#include <string>

using namespace std;

template<class T>
void Print(const T &container)
{
    for (const auto &y : container)
        cout << y << ' ';
    cout << endl;
}
//Функция для упорядочения целых чисел.
bool gt(int a,int b)
{
    int sa = 0;
    int sb = 0;
    while (a > 0) { sa += a % 10; a /= 10; }
    while (b > 0) { sb += b % 10; b /= 10; }
    return sa > sb;
}

int main()
{
    deque<int> d = { 8,21,123,321,9,12,15,51 };
    sort(d.begin(), d.end(), gt);
    Print(d);
    return 0;
}
```

```
9 8 123 321 15 51 21 12
```

Дело в том, что функциональные объекты могут работать только с базовыми типами C++ и с классами, для которых определены соответствующие операторы (+, -, <, == и т. д.). Это не всегда удобно, поскольку иногда приходится работать с типами данных, для которых такое условие не выполняется. В этом случае можно определить собственную пользовательскую функцию для функционального объекта. Например, оператор < не определен для обычных строк char*, но можно написать функцию, выполняющую сравнение, и использовать ее адрес (имя) вместо функционального объекта. Программа SORTCOM показывает, как отсортировать массив строк char*.

Листинг 7. Программа SORTCOM

```
//-----
// sortcom.cpp
```

```

// сортировка массива строк с помощью пользовательской
//функции сортировки
#include <vcl.h>
#include <iostream>
#include <string> // для strcmp()
#include <algorithm>
using namespace std;

// массив строк
char* names[] = { "Сергей", "Татьяна", "Елена",
                  "Дмитрий", "Михаил", "Владимир" };

bool alpha_comp(char*, char*); // объявление

int main()
{
    char* s;
    sort(names, names+6, alpha_comp); // сортировка строк

    for(int j=0; j<6; j++) // вывод отсортированных строк
    {
        s = names[j]; CharToOem(s,s);
        cout << s << endl;
    }
    int i;
    cin >> i;
    return 0;
}

bool alpha_comp(char* s1, char* s2) // возвращает true
// если s1<s2
{
    return ( strcmp(s1, s2)<0 ) ? true : false;
}
//-----

```

Третьим параметром алгоритма sort() в данном случае является адрес функции alpha_comp(), сравнивающей две строки типа char* и возвращающей true или false в зависимости от того, правильно ли для них осуществлена лексикографическая сортировка. Если неправильно, строки меняются местами. В программе используется библиотечная функция C под названием strcmp(), которая возвращает отрицательное значение в случае, если первый аргумент меньше второго. Результат программы вполне ожидаем:

Владимир Дмитрий Елена Михаил Сергей Татьяна

Вообще-то писать собственные функциональные объекты для работы с текстом вовсе не обязательно. Используйте класс string из стандартной библиотеки и будьте счастливы. Ибо в нем есть встроенные функциональные объекты, такие, как less<>() и greater<>().

Использование стандартных алгоритмов, как и других средств стандартной библиотеки, избавляет программиста от написания, отладки и документирования циклов обработки последовательностей, что уменьшает количество ошибок в программе, снижает время ее разработки и делает ее более читаемой и компактной.

Объявления стандартных алгоритмов находятся в заголовочном файле **<algorithm>**, стандартных функциональных объектов — в файле **<functional>**.

Все алгоритмы STL можно разделить на четыре категории:

- не модифицирующие операции с последовательностями;
- модифицирующие операции с последовательностями;
- алгоритмы, связанные с сортировкой;
- алгоритмы работы с множествами и пирамидами;

Кроме того, библиотека содержит обобщенные численные алгоритмы, объявления которых находятся в файле **<numeric>** «Средства для численных расчетов».

В качестве параметров алгоритму передаются итераторы, определяющие начало и конец обрабатываемой последовательности. Вид итераторов определяет типы контейнеров, для которых может использоваться данный алгоритм. Например, алгоритм сортировки (**sort**) требует для своей работы итераторы произвольного доступа, поэтому он не будет работать с контейнером **list**. Алгоритмы не выполняют проверку выхода за пределы последовательности.

Таблицы, приведенные в начале следующих разделов, дают представление о возможностях стандартных алгоритмов STL. Далее приведено описание каждого алгоритма. Следует учитывать, что для последовательностей, содержащих пользовательские типы данных, можно задавать собственные критерии.

При описании параметров шаблонов алгоритмов используются следующие сокращения:

In — итератор для чтения;

Out — итератор для записи;

For — прямой итератор;

Bi — двунаправленный итератор;

Ran — итератор произвольного доступа;

Pred — унарный предикат (условие);

BinPred — бинарный предикат;

Comp — функция сравнения;

Op — унарная операция;

BinOp — бинарная операция.

Не модифицирующие операции с последовательностями

Алгоритмы этой категории просматривают последовательность, не изменяя ее. Они используются для получения информации о последовательности или для определения положения элемента.

Таблица 1. Не модифицирующие операции с последовательностями

Алгоритм	Выполняемая функция
adjacent_find	Нахождение пары соседних одинаковых значений
count	Подсчет количества вхождений значения в последовательность
count_if	Подсчет количества выполнений условия в последовательности
equal	Попарное равенство элементов двух последовательностей
find	Нахождение первого вхождения значения в последовательность
find_end	Нахождение последнего вхождения одной последовательности в другую
find_first_of	Нахождение первого значения из одной последовательности в другой
find_if	Нахождение первого соответствия условию в последовательности
for_each	Вызов функции для каждого элемента последовательности
mismatch	Нахождение первого несовпадающего элемента в двух последовательностях
search	Нахождение первого вхождения одной последовательности в другую
search_n	Нахождение n-го вхождения одной последовательности в другую

Рассмотрим эти алгоритмы подробнее.

Алгоритм adjacent_find

Алгоритм adjacent_find выполняет нахождение пары соседних значений.

```
template<class For>
For adjacent_find(For first, For last);
template<class For, class BinPred>
For adjacent_find(For first, For last, BinPred pred);
```

Первая форма алгоритма находит в последовательном контейнере пару соседних одинаковых значений и возвращает итератор на первое из них или конец последовательности (итератор на элемент, следующий за последним).

Вторая форма находит соседние элементы, удовлетворяющие условию, заданному предикатом pred в виде функции или функционального объекта. Пример (программа находит самую левую пару одинаковых элементов целочисленного массива и пару элементов структуры, у которых равна сумма полей):

```
//-----
#include "stdafx.h"
#include <algorithm>
#include <iostream>
using namespace std;
//-----

struct A{ int x, y;};
bool f(A& a1, A& a2)
{return a1.x + a1.y == a2.x + a2.y;}
//-----

int _tmain(int argc, _TCHAR* argv[])
{
    int m[8] = {45, 60, 60, 25, 25, 2, 13, 35};
    cout << *(adjacent_find(m, m + 8)) << endl; //Output- 60

    A ma[5] = {{2,4}, {3,1}, {2,2}, {1,2}, {1,2}};
    cout << (*adjacent_find(ma, ma + 5, f)).x << endl; // Output-
3
    cin.get();
    return 0;
}
//-----
```

Алгоритм find()

Этот алгоритм ищет первый элемент в контейнере, значение которого равно указанному. В примере FIND показано, как нужно действовать, если мы хотим найти значение в массиве целых чисел.

Листинг 1. Программа FIND

```
//-----
// find.cpp
```

```

// найти первый объект, значение которого равно данному

#include "stdafx.h"
#include <algorithm>
#include <iostream>
#include <windows.h>
using namespace std;
//-----
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };

//-----

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    int* ptr;
    ptr = find(arr, arr+8, 33); //найти первое вхождение 33
    cout << "Первое вхождение значения 33 в позиции "
         << (ptr-arr) << endl;
    system("pause");
    return 0;
}
//-----

```

Результаты программы:

Первое вхождение значения 33 в позиции 2

Как и всегда, первый элемент в массиве имеет индекс 0, а не 1, поэтому число 33 было найдено в позиции 2, а не 3.

Поиск элемента в очереди с двусторонним доступом представлен программой ниже. Элемент 1 будет найден в очереди с двусторонним доступом d в позиции под номером 2.

```

//-----
//алгоритм find с deque
#include "stdafx.h"
#include <algorithm>
#include <deque>
#include <iostream>
#include <windows.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    int m[] = { 3, 5, 1, 8, 6 };
    deque<int> d(m,m + 5);
    deque<int>::iterator itr,it = d.begin();
    itr = find(d.begin(),d.end(),1); //найти первое вхождение 1
    cout << itr - it << endl; //будет выведено - 2. Это позиция 1.
}

```

```

    system("pause");
    return 0;
}
//-----

```

Поиск элемента в векторе представлен программой ниже. Она, по сути, ничем не отличается от предыдущей. Элемент 1 будет найден в векторе `v` в позиции под номером 2.

```

//-----
//алгоритм find с vector
#include "stdafx.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <windows.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    int m[] = { 3, 5, 1, 8, 6 };
    vector<int> v(m,m + 5);
    vector<int>::iterator itr, it = v.begin();
    itr = find(v.begin(), v.end(), 1);
    cout << itr - it << endl;//2
    system("pause");
    return 0;
}
//-----

```

Заголовочные файлы

В эту программу включен заголовочный файл `algorithm`. (Заметим, что, как и в других заголовочных файлах Стандартной библиотеки C++, расширения типа `.h` не пишут.) В этом файле содержатся объявления алгоритмов STL. Другие заголовочные файлы используются для контейнеров и для иных целей.

Диапазоны

Первые два аргумента алгоритма `find()` определяют диапазон просматриваемых элементов. Значения задаются итераторами. В данном примере мы использовали значения обычных указателей C++, которые, в общем-то, являются частным случаем итераторов.

Первый параметр — это итератор (то есть в данном случае — указатель) первого значения, которое нужно проверять. Вторым параметром — итератор последней позиции (на самом деле он указывает на следующую позицию за последним нужным значением). Так как всего в нашем массиве 8 элементов,

это значение равно первой позиции, увеличенной на 8. Это называется «значение после последнего».

Используемый при этом синтаксис является вариацией на тему обычного синтаксиса цикла for в C++:

```
for(int j = 0; j < 8; j++)    //от 0 до 7
{
    if(arr[j] == 33)
    {
        cout << " Первый объект со значением 33 найден в позиции " << j << endl;
        break; }
}
```

В программе FIND алгоритм find() не спасает вас от необходимости писать цикл for. В более интересных случаях, чем этот пример, алгоритмы могут спасти и не от такого. Очень сложные и длинные коды, бывает, заменяются алгоритмами.

Алгоритм count()

Взглянем на другой алгоритм — count(). Он подсчитывает, сколько элементов в контейнере имеют данное значение. В примере COUNT это продемонстрировано.

Листинг 2. Программа COUNT

```
//-----
// Алгоритм_count.cpp: определяет точку входа для консольного
// приложения.
//
// считает количество объектов, имеющих данное значение
#include "stdafx.h"
#include <iostream>
#include <algorithm>           //для count()
#include <string>
#include <windows.h>

using namespace std;
int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    string s1 = "Число 33 встречается ";
    string s2 = " раз(а) в массиве.";
    //Считать, сколько раз встречается 33.
    int n = count(arr, arr+8, 33);

    cout << s1 << n << s2 << endl;
```

```

    return 0;
}
//-----

```

На экране в результате мы увидим следующее:

Число 33 встречается 3 раз(а) в массиве.

Алгоритм sort()

Кажется, можно догадаться по названию, что делает этот алгоритм. Приведем пример применения его к массиву.

Листинг 3. Программа SORT

```

//-----
// sort.cpp
// сортирует массив целых чисел
#include "stdafx.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <windows.h>
using namespace std;
//-----
// массив чисел
int arr[] = {45, 2, 22, -17, 0, -30, 25, 55};

//-----

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    sort(arr, arr+8); // сортировка

    for(int j = 0; j < 8; j++) // вывести отсортированный
массив
        cout << arr[j] << ' ';
    cout << endl;
    system("pause");
    return 0;
}
//-----

```

Вот что мы видим на экране: -30. -17. 0. 2. 22. 25. 45. 55

К некоторым вариациям на тему этого алгоритма мы еще вернемся несколько позднее.

Алгоритм search()

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Например, если алгоритм find() ищет указанное значение в одном контейнере, алгоритм search() ищет целую последовательность значений, заданную одним контейнером, в другом контейнере. Рассмотрим на примере.

Листинг 4. Программа SEARCH

```
//-----  
// search.cpp  
//Ищем последовательность, заданную одним контейнером, в  
// другом контейнере  
#include "stdafx.h"  
#include <iostream>  
#include <algorithm>           //для search()  
#include <string>  
#include <windows.h>  
  
using namespace std;  
int source[] = { 11, 44, 33, 11, 22, 33, 11, 22, 44 };  
int pattern[] = { 11, 22, 33 };  
  
int _tmain(int argc, _TCHAR* argv[])  
{  
    setlocale(LC_ALL, "ru");  
    string s1 = "Совпадения не найдено\n";  
    string s2 = "Совпадение в позиции ";  
    int* ptr;  
    ptr = search(source, source+9, pattern, pattern+3);  
    if(ptr == source+9)// если после последнего  
        cout << s1;  
    else  
        cout << s2 << (ptr - source) << endl;  
    return 0;  
}  
//-----
```

Алгоритм просматривает последовательность 11, 22, 33, заданную массивом pattern, в массиве source. Видно, что эти числа подряд встречаются, начиная с четвертого элемента (позиция 3, считая с 0). Программа выводит на экран такой результат:

Совпадение в позиции 3

Если значение итератора ptr оказывается за пределами массива source, выводится сообщение о том, что совпадения не найдено.

Параметрами алгоритма search() и подобных не должны обязательно быть контейнеры одного типа. Исходный контейнер может быть, например, вектором STL, а маска поиска — обычным массивом. Такая универсальность — это одна из сильных сторон STL.

```
//-----  
#include "stdafx.h"  
#include <algorithm>  
#include <vector>  
#include <iostream>  
#include <windows.h>
```

```

#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    string s1 = "Совпадения не найдено\n";
    string s2 = "Совпадение в позиции ";
    int source[] = { 11, 44, 33, 11, 22, 33, 11, 22 };
    int pattern[] = { 11, 22, 33 };
    vector<int> v(source, source+8);
    vector<int>::iterator it;
    it = search(v.begin(), v.end(), pattern, pattern+3);
    if(it == v.end())// если после последнего
        cout << s1;
    else
        cout << s2 << (it - v.begin()) << endl;
    system("pause");
    return 0;
}
//-----

```

Алгоритм merge()

Этот алгоритм работает с тремя контейнерами, объединяя элементы двух из них в третий, целевой контейнер. Следующий пример показывает, как это делается.

Листинг 5. Программа MERGE

```

//-----
// merge.cpp
// соединение двух отсортированных контейнеров в третий
#include "stdafx.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <windows.h>
using namespace std;
//-----
int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
int dest[8];

//-----

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    //Соединить src1 и src2 в dest.
    merge(src1, src1 + 5, src2, src2 + 3, dest);
}

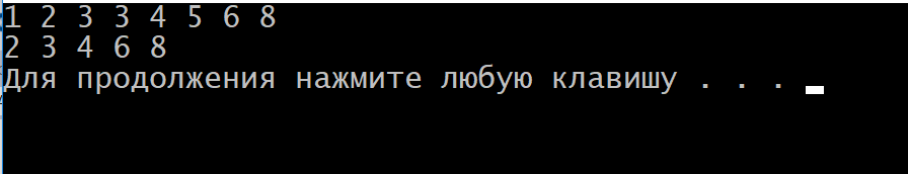
```

```

        // Вывести dest.
        for (const auto &y : dest)
            cout << y << ' ';
        cout << endl;
        // Вывести src1.
        for (const auto &y : src1)
            cout << y << ' ';
        cout << endl;
        system("pause");
        return 0;
    }
//-----

```

В итоге получается контейнер, содержимое которого таково:



```

1 2 3 3 4 5 6 8
2 3 4 6 8
Для продолжения нажмите любую клавишу . . . _

```

Как видите, алгоритм объединения сохраняет порядок следования элементов, вплетая содержимое двух контейнеров в третий. В следующем примере содержимое двух массивов объединяется в векторе.

```

//-----
#include "stdafx.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <windows.h>
using namespace std;
//-----
int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
vector<int> dest(8);

//-----

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "ru");
    //Соединить src1 и src2 в dest.
    merge(src1, src1+5, src2, src2+3, dest.begin());
    // Вывести dest.
    for (const auto &y : dest)
        cout << y << ' ';
    cout << endl;
    system("pause");
    return 0;
}
//-----

```

Добавление `_if` к аргументам

Некоторые аргументы имеют версии с окончанием `_if`. Им требуется дополнительный параметр, который называется предикатом и является функциональным объектом или функцией. Например, алгоритм `find()` находит все элементы, равные указанному значению. Можно написать функцию, которая работает с алгоритмом `find_if()` и находит элементы с какими-либо дополнительными параметрами.

В нашем примере используются объекты класса `string`. Алгоритму `find_if()` передается пользовательская функция `isDon()`, чтобы можно было искать первую строку в массиве, имеющую значение «Дмитрий». Приведем листинг примера.

Листинг 8. Программа FINDJF

```
//-----  
// find_if_deque.cpp: определяет точку входа для консольного  
// приложения.  
// Ищет в деке типа string первое вхождение слова «Дмитрий»  
#include "stdafx.h"  
#include <iostream>  
#include <string>  
#include <deque>  
#include <algorithm>  
using namespace std;  
//-----  
// Возвращает true, если name=="Дмитрий"  
bool isDon(string name)  
{  
    return name == "Дмитрий";  
}  
//-----  
int main()  
{  
    // Вызов функции настройки локали.  
    setlocale(LC_STYPE, "rus");  
    //Дека строк.  
    deque<string> names = { "Сергей", "Татьяна", "Елена", "Дмитрий",  
        "Михаил", "Владимир" };  
    //Итератор для дека строк.  
    deque<string>::iterator ptr;  
    string s1 = "Дмитрия нет в списке.\n";  
    string s2 = "Дмитрий записан в позиции ";  
  
    string s3 = " в списке.\n";  
    ptr = find_if(names.begin(), names.end(), isDon);  
  
    if (ptr == names.end())  
        cout << s1;
```

```

else
    cout << s2
        << (ptr - names.begin())
        << s3;
    cin.get();
    return 0;
}
//-----

```

Так как слово «Дмитрий» действительно встречается в деке, то результат работы программы будет такой:

```

Дмитрий записан в позиции 3 в списке.

```

Адрес функции isDon() — третий аргумент алгоритма find_if(), а первые два, как обычно, задают диапазон поиска от начала до «после последнего» элемента дека.

Алгоритм find_if() применяет функцию isDon() к каждому элементу из диапазона. Если isDon() возвращает true для какого-либо элемента, то find_if() возвращает значение итератора этого элемента. В противном случае возвращается указатель на адрес «после последнего» элемента дека.

Ниже приведён пример использования алгоритма find_if(), который ищет значения из заданного ему диапазона, с помощью функционального объекта D<>(). Функциональный объект задан шаблоном класса. И позволяет отыскивать элементы, значения которых лежат в указанном в реализации шаблона класса диапазоне. Так в примере отыскивается элемент вектора, значение которого находится в диапазоне(20..30). Таким значением в векторе будет 23.

```

//-----
#include "stdafx.h"
#include <algorithm> //для find
#include <fstream> //для работы с файлами
#include <iostream>
#include <vector>
using namespace std;

//Функциональный объект.
template<class T = int, int d = 0, int d1 = 50>
class Diapazon{
public:
    bool operator()(T x) {return x > d&& x < d1 ;};
};

int _tmain(int argc, _TCHAR* argv[])
{
    ifstream in("inpnum.txt");// f.txt
    vector<int> v;

```

```

    int x;
    while(in >> x,!in.eof())
        v.push_back(x);
    for(const auto &y : v)cout << y <<" ";//56 34 54 0 76 23 51 11 76
88
    cout << endl;
    cout << *find(v.begin(), v.end(), 51) << endl;// 51
    cout << *find_if(v.begin(), v.end(), Diapazon<int, 20, 30>()) <<
endl;// 23
    cin.get();
    return 0;
}
//-----

```

`_if`-версии имеются и у других алгоритмов, например `count()`, `replace()`, `remove()`.

Алгоритм `for_each()`

Этот алгоритм позволяет выполнять некое действие над каждым элементом в контейнере. Вы пишете собственную функцию, чтобы определить, какое именно действие следует выполнять. Эта ваша функция не имеет права модифицировать данные, но она может их выводить или использовать их значения в своей работе. Вот пример, в котором `for_each()` используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран. Мы пишем функцию `in_to_cm()`, которая просто умножает значение на 2,54, и передаем адрес этой функции в качестве третьего параметра алгоритма.

Листинг 9. Программа FOR_EACH

```

//-----
// for_each() используется для вывода элементов массива,
//переведенных из дюймов в сантиметры
#include "stdafx.h"
#include <iostream>
#include <algorithm>
using namespace std;
// Объявление функции.
void in_to_cm(double);

int _tmain(int argc, _TCHAR* argv[])
{
    // Массив значений в дюймах.
    double inches[] = {3.5, 6.2, 1.0, 12.75, 4.33};
    // Вывод в виде сантиметров.
    for_each(inches, inches+5, in_to_cm);
    cout << endl;
    cin.get();
    return 0;
}

```



```
// Определение функции: перевод и вывод в сантиметрах.
void in_to_cm(double in)
{
    cout << (in * 2.54) << ' ';
}
//-----
```

Результаты программы выглядят так:

8.89 15.748 2.54 32.385 10.9982

Алгоритм transform()

Этот алгоритм тоже делает что-то с каждым элементом контейнера, но еще и помещает результат в другой контейнер (или в тот же). Опять же, пользовательская функция определяет, что именно делать с данными, причем тип возвращаемого ею результата должен соответствовать типу целевого контейнера. Наш пример, в общем-то, идентичен примеру FOR_EACH за единственным исключением: вместо вывода на экран, функция in_to_cm() выводит значения сантиметров в новый массив centi[], затем главная программа выводит содержимое этого массива. Приведем листинг программы.

Листинг 10. Программа TRANSFO

```
//-----
// transfo.cpp
// transform() используется для перевода значений из дюймов //в
сантиметры
#include "stdafx.h"
#include <iostream>
#include <algorithm>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    //Массив дюймов.
    double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
    //Массив сантиметров.
    double centi[5];
    // Прототип функции преобразования дюймов в сантиметры.
    double in_to_cm(double); // прототип
    // перевод в массив centi[]
    transform(inches, inches + 5, centi, in_to_cm);
    // вывод массива centi[]
    for (const auto &y : centi)
        cout << y << ' ';
    cout << endl;
    cin.get();
    return 0;
}
// Перевод дюймов в сантиметры.
```

```
double in_to_cm(double in)
{
    return (in * 2.54);    // вернуть результат
}
//-----
```

Результат работы программы:

```
8.89 15.748 2.54 32.385 10.9982
-
```

В программе, приведённой ниже, выполняются те же преобразования данных, только исходные данные в дюймах берутся из массива, а результат в сантиметрах записывается в список. Из него данные и выводятся на экран.

```
//-----
// transform() используется для перевода значений из дюймов в
сантиметры
#include "stdafx.h"
#include <list>
#include <iostream>
#include <algorithm> //для transform
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // Массив дюймов.
    double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
    //Список сантиметров.
    list<double> centi(5);
    // Прототип функции преобразования дюймов в сантиметры.
    double in_to_cm(double);
    // Перевод в список centi[]
    transform(inches, inches + 5, centi.begin(), in_to_cm);
    list<double>::iterator it;
    // Вывод списка centi[].
    for (const auto &y : centi)
        cout << y << ' ';
    cout << endl;
    int i;
    cin >> i;
    return 0;
}
// Перевод дюймов в сантиметры.
double in_to_cm(double in)
{
    return (in * 2.54);    // вернуть результат
}
//-----
```

Результат работы программы:

```
8.89 15.748 2.54 32.385 10.9982
```

В этом примере находится скользящее среднее значений двух векторов и результат записывается в третий вектор.

```
#include "stdafx.h"
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

template<class T>
void Print(const T &container)
{
    cout << '(' ;
    for (const auto &y : container)
        cout << y << ' ';
    cout << ')' << endl;
}

template <class Type>
class average
{
public:
    Type operator( ) (Type a,
                     Type b)
    {
        return (Type)((a + b) / 2);
    }
};

int main()
{
    vector <double> v1, v2, v3(6);

    for (int i = 1; i <= 6; i++)
        v1.push_back(11.0 / i);

    for (int j = 0; j <= 5; j++)
        v2.push_back(-2.0 * j);

    cout << "The vector v1 = ";
    Print(v1);

    cout << "The vector v2 = ";
    Print(v2);

    // Finding the element-wise averages of the elements of v1 & v2
    transform(v1.begin(), v1.end(), v2.begin(), v3.begin(),
              average<double>());

    cout << "The element-wise averages are: ";
    Print(v3);
}
```

Результат работы программы:

```
The vector v1 = (11 5.5 3.66667 2.75 2.2 1.83333 )
The vector v2 = (-0 -2 -4 -6 -8 -10 )
The element-wise averages are: (5.5 1.75 -0.166667 -1.625 -2.9 -4.08333 )
Для продолжения нажмите любую клавишу . . . █
```

Пример использования алгоритма `for_each` для умножения элементов вектора на константу и вычисления среднего значения.

```
#include "stdafx.h"
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

template<class T>
void Print(const T &container)
{
    cout << '(';
    for (const auto &y : container)
        cout << y << ' ';
    cout << ')';
    cout << endl;
}

// Функциональный объект для умножения элементов на фактор.
template <class Type>
class MultValue
{
private:
    Type Factor;    // Значение на которое умножаем.
public:
    // Конструктор.
    MultValue(const Type& value) : Factor(value) {
    }

    // Вызов функции для умножаемого элемента.
    void operator( ) (Type& elem) const
    {
        elem *= Factor;
    }
};

// Функциональный элемент для нахождения среднего.
class Average
{
private:
    long num;        // Число элементов.
    long sum;        // Сумма элементов.
public:
    // Конструктор.
    Average() : num(0), sum(0)
    {
    }
};
```

```

    }

    // Функция для обработки следующего элемента.
    void operator( ) (int elem)
    {
        num++;          // Инкремент счётчика.
        sum += elem;    // Накопление суммы.
    }

    // Возвращает среднее.
    operator double()
    {
        return static_cast<double> (sum) /
               static_cast<double> (num);
    }
};

int main()
{
    setlocale(LC_ALL, "ru");
    vector<int> v1;
    vector<int>::iterator Iter1;

    // Конструируем вектор v1.
    int i;
    for (i = -4; i <= 2; i++)
    {
        v1.push_back(i);
    }

    cout << "Исходный вектор v1 = ";
    Print(v1);

    // Используем for_each для умножения каждого элемента на Factor.
    for_each(v1.begin(), v1.end(), MultValue<int>(-2));

    cout << "Умножаем элементы вектора v1\n"
         << "на factor -2 получаем:\n v1mod1 = ";
    Print(v1);

    // Функциональный объект настроен и так его можно использовать
    // для элементов с различными значениями Factor.
    for_each(v1.begin(), v1.end(), MultValue<int>(5));

    cout << "Умножаем элементы вектора v1mod\n"
         << "на factor 5 получаем:\n v1mod1 = ";
    Print(v1);

    // Состояние функционального объекта может накапливать
    // информацию о последовательных действиях чтобы вернуть
    значение Average

```

```

    double avemod2 = for_each(v1.begin(), v1.end(),
        Average());
    cout << "Среднее значение элементов вектора is:\n Average (
v1mod2 ) = "
        << avemod2 << "." << endl;
}

```

```

Исходный вектор v1 = (-4 -3 -2 -1 0 1 2 )
Умножаем элементы вектора v1
на factor -2 получаем:
v1mod1 = (8 6 4 2 0 -2 -4 )
Умножаем элементы вектора v1mod
на factor 5 получаем:
v1mod1 = (40 30 20 10 0 -10 -20 )
Среднее значение элементов вектора is:
Average ( v1mod2 ) = 10.
Для продолжения нажмите любую клавишу . . .

```

Итак, мы рассмотрели некоторые алгоритмы STL. Есть еще множество других, но, кажется, из приведенных примеров вполне понятно, какие они бывают и как с ними следует обращаться.