контейнеры и итераторы лекция

<http://www.amse.ru/courses/cpp2/2011_02_21.html>

Алгоритмы min\_element() и max\_element() находят минимальный и максимальный элементы в контейнере:



|  |  |
| --- | --- |
|  | #include <iostream>  #include <list>  #include <algorithm>    int main()  {      std::list<int> li;      for (int nCount=0; nCount < 5; ++nCount)          li.push\_back(nCount);        std::list<int>::const\_iterator it; // объявляем итератор      it = min\_element(li.begin(), li.end());          std::cout << \*it << ' ';      it = max\_element(li.begin(), li.end());          std::cout << \*it << ' ';        std::cout << '\n';  } |

Результат выполнения программы:

0 4

**Алгоритмы find() и list::insert()**

В следующем примере мы используем алгоритм find(), чтобы найти определенное значение в списке, а затем используем функцию list::insert() для добавления нового значения в список:



|  |  |
| --- | --- |
|  | #include <iostream>  #include <list>  #include <algorithm>    int main()  {      std::list<int> li;      for (int nCount=0; nCount < 5; ++nCount)          li.push\_back(nCount);        std::list<int>::iterator it; // объявляем итератор      it = find(li.begin(), li.end(), 2); // ищем в списке число 2      li.insert(it, 7); // используем алгоритм list::insert() для добавления числа 7 перед числом 2        for (it = li.begin(); it != li.end(); ++it) // выводим с помощью цикла и итератора элементы списка          std::cout << \*it << ' ';        std::cout << '\n';  } |

Результат выполнения программы:

0 1 7 2 3 4

В следующем примере мы отсортируем весь вектор, выведем отсортированные элементы, а затем выведем их в обратном порядке:



|  |  |
| --- | --- |
|  | #include <iostream>  #include <vector>  #include <algorithm>    int main()  {        std::vector<int> vect;      vect.push\_back(4);      vect.push\_back(8);      vect.push\_back(-3);      vect.push\_back(3);      vect.push\_back(-8);      vect.push\_back(12);      vect.push\_back(5);        sort(vect.begin(), vect.end()); // выполняем сортировку элементов вектора        vector<int>::const\_iterator it; // объявляем итератор      for (it = vect.begin(); it != vect.end(); ++it) // выводим с помощью цикла и итератора элементы вектора          cout << \*it << ' ';       cout << '\n';        std::reverse(vect.begin(), vect.end()); // сортируем элементы вектора в обратную сторону        for (it = vect.begin(); it != vect.end(); ++it) // выводим с помощью цикла и итератора элементы вектора          std::cout << \*it << ' ';        std::cout << '\n';  } |

Результат выполнения программы:

-8 -3 3 4 5 8 12  
12 8 5 4 3 -3 -8

Обратите внимание, общий алгоритм sort() не работает с вектором, у вектора есть свой собственный метод sort(), который, в данном случае, является более эффективным.

 Выполнение пользовательской функции (в данном случае, вывода в консоль) для каждого элемента вектора.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

void myfunction (int i) { cout << ' ' << i; }

int main () {

vector<int> myvector;

myvector.push\_back(10);

myvector.push\_back(20);

myvector.push\_back(30);

for\_each (myvector.begin(), myvector.end(), myfunction);

return 0;

}

Пример 9. Поиск элемента в целочисленном векторе.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main () {

int myints[] = { 10, 20, 30, 40 };

vector <int> myvector (myints,myints+4);

vector<int>::iterator it;

it = find (myvector.begin(), myvector.end(), 30);

if (it != myvector.end())

cout << "Element found in myvector: " << \*it << '\n';

else

cout << "Element not found in myvector\n";

cin.get(); return 0;

}

Пример 10. Замена нечётных значений целочисленного вектора нулями.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

bool IsOdd(int i) { return ((i%2)==1); } //Проверка на нечётность

int main () {

vector<int> myvector;

for (int i=1; i<10; i++) myvector.push\_back(i);

replace\_if (myvector.begin(), myvector.end(), IsOdd, 0);

for (vector<int>::iterator it=myvector.begin(); it!=myvector.end(); it++)

cout << \*it << ' ';

cin.get(); return 0;

}

Пример 11. Сортировка элементов целочисленного вектора по убыванию

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

bool myfunction (int i,int j) { return (i>j); }

int main () {

int myints[] = {32,71,12,45,26,80,53,32};

vector<int> myvector (myints, myints+8);

sort (myvector.begin(), myvector.end(), myfunction);

for (vector<int>::iterator it=myvector.begin(); it!=myvector.end(); it++)

cout << \*it << ' ';

cin.get(); return 0;

}

Пример 12. Реализация объединения двух целочисленных множеств.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main () {

int first[] = {25,10,15,20,5};

int second[] = {50,40,30,20,10};

sort (first,first+5);

sort (second,second+5);

//set\_union сработает для отсортированных данных!

vector <int> v(10);

vector <int>::iterator it=

set\_union (first, first+5, second, second+5, v.begin());

v.resize(it-v.begin());

for (it=v.begin(); it!=v.end(); it++) cout << \*it << ' ';

cin.get(); return 0;

}

Пример 13. Поиск строки с наименьшей длиной в тексте.

#include <iostream>

#include <string>

#include <algorithm>

#include <vector>

using namespace std;

bool myMin (string a, string b) { return a.length()<b.length(); }

int main () {

string text[] = {

"Mama", "pomila", "ramochku", "milom", ",", "yeah", "!"

};

vector <string> v(text,text+7);

vector <string>::iterator it=min\_element(v.begin(),v.end(),myMin);

cout << \*it;

cin.get(); return 0;

}

Пример 14. Вывод всех перестановок элементов целочисленного массива.

#include <iostream>

#include <algorithm>

using namespace std;

int main () {

int a[] = { 1, 2, 3, 4 };

int len = sizeof(a) / sizeof(int);

do {

for (int i = 0; i < len; i++) cout << a[i] << ' ';

cout << endl;

} while (next\_permutation (a, a + len));

cin.get(); return 0;

}

Пример 15. Сгенерировать массив случайных чисел, равномерно распределённых на заданном интервале [a,b]. В новых стандартах C++ есть библиотека <random> со всеми распределениями. Распространённый подход rand()%число как раз вряд ли даст равномерное распределение из-за взятия операции "остаток от деления" %.

#include <iostream>

#include <random>

using namespace std;

int main() {

const int n = 100; //количество значений

const int a = 1; //границы интервала

const int b = 1000;

//опишем генератор и массив

default\_random\_engine generator;

uniform\_int\_distribution<int> distribution(a,b);

int p[n]={};

//заполнение и вывод

for (int i=0; i<n; ++i) {

p[i] = distribution(generator);

cout << p[i] << " ";

}

cin.get(); return 0;

}

Пример 16. Из контейнера комплексных чисел удалить значения, мнимая часть которых равна нулю.

#include <iostream>

#include <algorithm>

#include <iterator>

#include <string>

#include <list>

using namespace std;

struct complex {

double re,im;

};

template <typename T> void vType (T v) {

cout << endl;

for (auto p=v.begin(); p<v.end(); p++) cout << \*p << " ";

}

bool filter (complex c) { return c.im==0; }

int main() {

list <complex> q;

cout << "Type data, 0 0 is exit";

complex n;

int i=1;

while (1) {

cout << endl << i++ << ": ";

cin >> n.re >> n.im;

if (n.re==0 && n.im==0) break;

q.push\_back(n);

}

list <complex> q2;

remove\_copy\_if (q.begin(),q.end(),back\_inserter(q2),filter);

//back\_inserter обеспечивает вставку результатов

//работы алгоритма в новый контейнер

list <complex>::iterator p;

for (p=q2.begin(); p!=q2.end(); p++)

cout << '(' << (p)->re << ',' << (p)->im << ')' << endl;

cin.sync(); cin.get(); return 0;

}

Все коды проверены в Visual C++ Express 2010, с точностью до [\_CRT\_SECURE\_NO\_WARNINGS](http://blog.kislenko.net/show.php?id=1401#6602) должны работать и в версиях постарше.

Подходящая задача по теме - "сделать что-нибудь с помощью контейнеров, алгоритмов и итераторов", наподобие последнего примера.

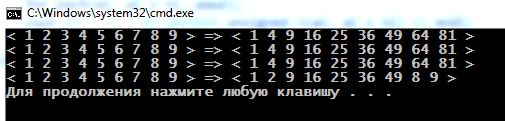
Контейнеры STL представляли бы собой красивую выдумку достаточно далёкую от практического использования (как и было в первые годы существования STL), если бы не следующее обстоятельство: из-за единой **общей** природы всех контейнеров основные алгоритмы, представляющие интерес на практике, могут быть реализованы в обобщённом виде, применимом к **любым** типам контейнеров.

Алгоритмы — это самая объёмная и самая востребованная часть библиотеки. Предоставляется настолько много алгоритмов, что для детального описания их всех не хватит и объёмной книги. Ниже мы совершенно **условно** поделим их на группы и назовём по именам (и тоже далеко не все), и только по некоторым построим примеры использования.

**for\_each()** Наиболее часто используемый алгоритм — это **for\_each()**: выполнение действия для группы элементов (возможно всех) контейнера. Ниже показано несколько примеров работы алгоритма **for\_each()** для массива и вектора, точно также этот алгоритм может использоваться с **любым** контейнером STL:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #include <iostream>  #include <vector>  #include <algorithm>    using namespace std;    inline ostream& operator <<( ostream& out, const vector< unsigned > & obj ) {     cout << "< ";     for( auto& p: obj )        cout << p << " ";     return out << ">";  }    void pow2( unsigned& i ) { i \*= i; }    int main( void ) {     const int examples = 4;     for( int i = 0; i < examples; i++ ) {        unsigned ai[] = { 1, 2, 3, 4 , 5, 6, 7, 8, 9 },             ni = sizeof( ai ) / sizeof( ai[ 0 ] );        vector< unsigned > vi( ai, ai + ni );        cout << vi;        switch( i ) {           case 0:              for\_each( vi.begin(), vi.end(), pow2 );              cout << " => " << vi << endl;              break;           case 1:              for\_each( ai, ai + ni, pow2 );              cout << " => " << vector< unsigned >( ai, ai + ni ) << endl;              break;           case 2:              for( auto& i : ai ) pow2( i );              cout << " => " << vector< unsigned >( ai, ai + ni ) << endl;              break;           case 3:              for\_each( vi.begin() + 2, vi.end() - 2, pow2 );              cout << " => " << vi << endl;              break;        }     }  } |

[](https://purecodecpp.com/wp-content/uploads/2016/04/algoritmy-stl.jpg)

**Примечание:** Строкой 3 показана работа новой (введена стандартом C++11) конструкции for( auto &x : …), которая имеет подобный эффект присвоения и может применяться и к массивам и к контейнерам (в функции-операторе вывода вектора в поток показан такой вариант). Эта конструкция, вообще то говоря, не является составной частью библиотеки STL или алгоритмов, но имеет тот же эффект, что и алгоритм for\_each(): применить **последовательно ко всем элементам коллекции**.

Этот пример показывает основную логику организации **всех** алгоритмов: к указанному диапазону (не обязательно ко всей коллекции), ограниченному итератором начала и конца (зачастую указываемых первыми 2-мя параметрами) применяется поочерёдно функция, функтор, или предикат (функция, возвращающая логиxческий результат, позволяющий произвести отбор по какому-либо признаку).

**find()** Следующий по значимости алгоритм — это **find()**. Как интуитивно понятно из имени, это поиск некоторого элемента в коллекции. Обратите внимание, что многие контейнеры имеют **метод** **find()**, который для объекта будет вызываться как **obj.find(…)**, в то время как **алгоритм** будет вызываться как **find( obj:iteator, … )**.

Собственно, это даже не один этот алгоритм, а целая обширная их группа, которую можно объединить по признаку того, что они **отбирают** элементы коллекции по какому-то признаку, условию, [предикату](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%B8%D0%BA%D0%B0%D1%82" \t "_blank): **find()**, **find\_if()**, **find\_if\_not()**, **find\_first\_of()**, **find\_end()**, **adjacent\_find()**. В эту же группу, с некоторой натяжкой, можно отнести и **count()**, **count\_if()**, **search()**, **binary\_search()**, **min()**, **max()**, **minmax\_element()**, **min\_element()**, **max\_element()**, **equal()** и др.

Ещё одна условная группа — это алгоритмы, некоторым образом «тасующие» коллекцию, **переставляющие** элементы местами, меняющие значения: **fill()**, **replace\_copy()**, **reverse()**, **rotate()**, **rotate\_copy()**, **shuffle()**, **random\_shuffle()**, **transform()**, **replace()**, **replace\_if()** и др.

Ещё группа — это алгоритмы работающие с 2-мя коллекциями, **копирующие и перемещающие** содержимое (причём, возможно, между коллекциями разного вида, например, **vector<>** в **set<>**): **copy()**, **copy\_if()**, **move()**, **swap\_ranges()**, **remove\_copy()**, **remove\_copy\_if()**, **merge()**,**set\_intersection()**, **set\_difference()** и др.

И, наконец, совсем особая группа алгоритмов связана с разнообразными **сортировками** элементов внутри коллекции: **sort()**, **stable\_sort()**, **is\_sorted()**, **is\_sorted\_until()** и др. Эту интересную группу мы отложим на потом, для отдельного обстоятельного рассмотрения.

При таком обилии реализованных алгоритмов и число которых в библиотеке со временем возрастает, и при том, что большинство из них вообще толком нигде не описаны в литературе, возникает естественный вопрос: как разобраться во всём этом разнообразии? Эти сложности снимаются тем что:

* Все объекты STL (контейнеры, алгоритмы) описаны в синтаксисе шаблонов (**template**). Поэтому их описания **обязательно** должны включаться в компилируемый код в составе своих заголовочных файлов (хедер-файлов).
* Отправляйтесь в стандартный каталог **</usr/include/c++>** и найдите там хедер-файлы файлы вида **stl\_algo\*** — в них вы найдёте все прототипы функций алгоритмов. Более того, там же каждому прототипу предшествует обстоятельный **комментарий**, объясняющий назначение алгоритма, и объясняющий параметры вызова.
* Рассмотрите примеры кода, использующие несколько **основных** алгоритмов STL — в сети их множество. По аналогии элементарно просто воспроизвести поведение и **всех** остальных алгоритмов.

**Примечание:** Вот из-за того, что библиотеки шаблонных классов определены в терминах **template**, сообщения об синтаксических ошибках компиляции становятся а). многословными, на десятки строк сообщений и б). ужасно невнятными для поиска ошибок. Это оборотная сторона медали такого мощного механизма как **template**, и к этому нужно быть готовым.

Как и было сказано выше, изучение примеров снимет множество вопросов, а поэтому приступим к коду … Теперь внимательно следите за руками (алгоритмов STL такое множество, что комментировать каждый – за пределами разумного объёма изложения, но все они работают **подобно** один другому):



аны контейнеры для **char** (как компактные, но самые неприятные в работе), над которыми выполняются разнообразные алгоритмы практически всех обозначенных групп: