Лекции по C/C++: Стандартная библиотека шаблонов (STL)

> Капитан Очевидность подсказывает:
> STL - это способ программировать, не уча типовые алгоритмы

STL – стандартная библиотека шаблонов, включённая в новые версии стандарта C++. STL обеспечивает стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных. STL строится на основе *шаблонов классов*, поэтому входящие в неё алгоритмы и структуры применимы почти ко всем типам данных. Ядро библиотеки образуют три элемента: контейнеры, алгоритмы и итераторы.

- *Контейнеры* (containers) - это объекты, предназначенные для хранения набора элементов. Например, вектор, линейный список, множество. Ассоциативные контейнеры (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям. В каждом классе-контейнере определен набор функций для работы с ними. Например, контейнер "список" содержит функции для вставки, удаления и слияния элементов.
- *Алгоритмы* (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.
- *Итераторы* (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера и сканировать его элементы, примерно так же, как указатели используются для доступа к элементам массива. С итераторами можно работать так же, как с указателями. К ним можно применять операции *, инкремент, декремент. Тип итератора iterator определён в различных контейнерах.

Существует пять типов итераторов:

1. Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента: ==, !=, *i, ++i, i++, *i++. Специальным случаем итератора ввода является istream_iterator.

2. Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: ++i, i++, *i = t, *i++ = t. Специальным случаем итератора вывода является ostream_iterator.

3. Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание: ==, !=, =, *i, ++i, i++, *i.

4. Двунаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, *i--), что позволяет им проходить контейнер в обоих направлениях.

5. Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ к элементу по индексу: i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2.

В STL также поддерживаются обратные итераторы (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, проходящие последовательность в обратном направлении.

Кроме контейнеров, алгоритмов и итераторов, в STL поддерживается еще несколько стандартных компонентов.

Главными среди них являются распределители памяти, предикаты и функции сравнения.

У каждого контейнера имеется определенный для него распределитель памяти (allocator), который управляет процессом выделения памяти для контейнера. По умолчанию распределителем памяти является объект класса allocator. Можно определить собственный распределитель.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая предикатом. Предикат может быть унарным и бинарным. Возвращаемое значение предиката - истина или ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов UnPred, бинарных - BinPred. Тип аргументов соответствует типу хранящихся в контейнере объектов.

Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется функцией сравнения (comparison function). Функция возвращает истину, если первый элемент меньше второго. Типом функции является тип Comp:

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

Особую роль в STL играют объекты-функции (функторы). Объекты-функции - это экземпляры класса, в котором определена операция "круглые скобки" (). В ряде случаев удобно заменить функцию на объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется operator(). Приведём законченный пример кода, реализующего данный подход.

Пример 1. Удаление строк из списка с помощью функтора.

```
#include <iostream>
#include <functional>
#include <iterator>
#include <string>
#include <list>
using namespace std;

class contains : public unary_function <string, bool> {
 //наличие подстроки в строке
 private:
  string substr_;
 public:
  contains(const string & substr) : substr_(substr) { } //конструктор
  bool operator () (const string& s) const {
   //благодаря перегрузке этого оператора,
```

```
   //объект класса может "прикидываться" функцией
   return s.find(substr_) != string::npos;
  }
};

int main() {
 typedef ostream_iterator <string> string_ostream_iter_t;
 typedef list <string> string_list_t;

 string_list_t lst;
 lst.push_back("one"); //заполняем список
 lst.push_back("two");
 lst.push_back("three");
 lst.push_back("four");
 lst.push_back("five");

 lst.remove_if(contains("o")); //удаляем строки содержащие букву "o"
 lst.remove_if(not1(contains("r")));
  //а так можно удалить строки, не содержащие букву "r"

 // выводим оставшиеся элементы списка на экран
 copy (lst.begin(), lst.end(), string_ostream_iter_t(cout, "\n"));
 cin.get(); return 0;
}
```

Здесь использованы имеющиеся в STL метод push_back и алгоритмы remove_if, copy, они описаны ниже по тексту.

В STL определены два основных типа контейнеров: последовательности и ассоциативные контейнеры. Разница между ними состоит в том, что для последовательностей имеет значение порядок следования элементов (вектор, стек, очередь), а для ассоциативных контейнеров – нет (ассоциативный массив, множество).

Ключевая идея для стандартных контейнеров заключается в том, что они должны быть взаимозаменяемыми, если это разумно для решения задачи. Пользователь может выбирать между контейнерами, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться ассоциативным массивом map. С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться списочным контейнером list. Если добавление и удаление элементов обычно производится в начало или конец контейнера, уместно применение очереди queue, двусторонней очереди deque или стека stack. По умолчанию пользователь обычно применяет контейнер vector, он реализован, чтобы хорошо работать для широкого диапазона задач.

Идея обращения с различными видами контейнеров унифицированным способом ведёт к понятию обобщённого программирования. Для поддержки этой идеи STL содержит множество готовых обобщённых алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности внутреннего устройства отдельных контейнеров.

Кратко опишем основные возможности STL, подробнее с ними можно познакомиться в стандартной справке или на сайте cplusplus.com.

Классы-контейнеры STL

Наименование класса	Описание	Заголовочный файл
bitset	множество битов	**<bitset>**
vector	динамический массив	**<vector>**
list	линейный список	**<list>**
deque	двусторонняя очередь	**<deque>**
stack	стек	**<stack>**
queue	очередь	**<queue>**
priority_queue	очередь с приоритетом	**<queue>**
map	ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение	**<map>**
multimap	с каждым ключом связано два или более значений	**<map>**
set	множество	**<set>**
multiset	множество, в котором каждый элемент не обязательно уникален	**<set>**

Типы STL

Идентификатор типа	Описание
value_type	тип элемента
allocator_type	тип распределителя памяти
size_type	тип индексов, счетчика элементов и т.д.
iterator	ведёт себя как **value_type ***
reverse_iterator	просматривает контейнер в обратном порядке
reference	ведёт себя как **value_type &**
key_type	тип ключа (только для ассоциативных контейнеров)
key_compare	тип критерия сравнения (только для ассоциативных контейнеров)
mapped_type	тип отображенного значения

Итераторы STL

Итератор	Описание
begin()	указывает на первый элемент
end()	указывает на элемент, следующий за последним
rbegin()	указывает на первый элемент в обратной последовательности
rend()	указывает на элемент, следующий за последним в обратной последовательности

Методы доступа к элементам STL

Метод	Описание
front()	ссылка на первый элемент
back()	ссылка на последний элемент
operator [](i)	доступ по индексу без проверки
at(i)	доступ по индексу с проверкой

Методы для включения и исключения элементов

Метод	Описание
insert(p, x)	добавление **x** перед элементом, на который указывает **p**
insert(p, n, x)	добавление **n** копий **x** перед **p**
insert(p, first, last)	добавление элементов из диапазона **[first:last)** перед **p**
push_back(x)	добавление **x** в конец
push_front(x)	добавление нового первого элемента (только для списков и очередей с двумя концами)
pop_back()	удаление последнего элемента
pop_front()	удаление первого элемента (только для списков и очередей с двумя концами)
erase(p)	удаление элемента в позиции **p**
erase(first, last)	удаление элементов из диапазона **[first:last)**
clear()	удаление всех элементов

Другие операции STL

Операция	Описание
size()	количество элементов
empty()	определяет, пуст ли контейнер
capacity()	память, выделенная под вектор (только для векторов)
reserve(n)	выделяет память для контейнера под n элементов
resize(n)	изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
swap(x)	обмен местами двух контейнеров
==, !=, <	операции сравнения
operator =(x)	контейнеру присваиваются элементы контейнера **x**
assign(n, x)	присваивание контейнеру **n** копий элементов **x** (не для ассоциативных контейнеров)
assign(first, last)	присваивание элементов из диапазона **[first:last)**
operator [](k)	доступ к элементу с ключом **k**
find(k)	находит элемент с ключом **k**
lower_bound(k)	находит первый элемент с ключом, меньшим **k**
upper_bound(k)	находит первый элемент с ключом, большим **k**
equal_range(k)	находит **lower_bound** (нижнюю границу) и **upper_bound** (верхнюю границу) элементов с ключом **k**

Вектор vector в STL определен как динамический массив с доступом к его элементам по индексу.

```
template <class T, class Allocator = allocator<T> > class std::vector
{
// ...
};
```

где T - тип предназначенных для хранения данных. Allocator задает распределитель памяти, который по умолчанию является стандартным. В классе vector определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());
 // конструктор пустого вектора
explicit vector(size_type число, const T &значение = T(),
                const Allocator &a = Allocator());
 //конструктор с заданным количеством и значением элементов
vector(const vector<T, Allocator> &объект);
 //конструктор копирования
template<class InIter> vector(InIter начало, InIter конец,
                              const Allocator &a = Allocator());
 //конструктор вектора, содержащего диапазон элементов,
 //заданный итераторами "начало" и "конец"
```

Примеры вызова этих конструкторов:

```
vector <int> a;
vector <double> x(5);
vector <char> c(5, '*');
vector <int> b(a); // b = a
```

Общие принципы работы с вектором следующие:

- Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и ==.
- Для класса вектор определены операторы сравнения ==, <, <=, !=, >, >=
- Кроме этого, для класса vector определяется оператор индекса []
- Новые элементы могут включаться с помощью функций insert(), push_back(), resize(), assign()
- Существующие элементы могут удаляться с помощью функций erase(), pop_back(), resize(), clear()
- Доступ к отдельным элементам осуществляется с помощью итераторов begin(), end(), rbegin(), rend()
- Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций, подключаемых заголовочным файлом <algorithm>

Приведём пример кода, показывающего работу с вектором целых чисел.

Пример 2. Формирование и вывод вектора целых значений.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
template <typename T> void vType (T v) {
 int n = v.size();
 cout << endl;
 for (int i=0; i<n; i++) cout << v[i] << " ";
}

int main(void) {
 vector <int> v;
 for (int i = 0; i < 10; i++) v.push_back(i+1);
 cout << "size = " << v.size() << endl;
 vType (v); //вывели полученный вектор
 for (int i = 0; i < 10; i++) v[i] = v[i] + v[i];
 vType (v); //вывели вектор с удвоенными значениями элементов
 cin.sync(); cin.get(); return 0;
}
```

В показанном ниже коде для последовательного доступа к элементам вектора используется итератор.

Пример 3. Доступ к элементам вектора с помощью итератора.

```
#include <iostream>
#include <vector>
using namespace std;

int main(void) {
 vector <int> v;
 for (int i = 0; i < 10; i++) v.push_back(i+1);

 vector <int>::iterator p = v.begin();
 while (p != v.end()) cout << *p++ << " ";

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

Основной цикл вывода элементов вектора, разумеется, мог иметь вид

```
vector <int>::iterator p;
for (p=v.begin(); p<v.end(); p++) cout << *p << " ";
```

Вывести элементы вектора в обратном порядке можно было кодом

```
vector <int>::reverse_iterator p;
for (p=v.rbegin(); p<v.rend(); p++) cout << *p << " ";
```

Следующий код демонстрирует вставку и удаление элементов вектора.

Пример 4. Добавление и удаление элементов вектора средствами STL.

```
#include <iostream>
#include <vector>
using namespace std;

template <typename T> void vType (T v, char *msg) {
```

```
 cout << endl << msg << ": ";
 for (auto p=v.begin(); p<v.end(); p++) cout << *p << " ";
}

int main (void) {
 vector <int> v(5, 1);
 vType (v,"Start vector");

 vector<int>::iterator p = v.begin();
 p += 2;
 v.insert(p, 10, 5); //вставить 10 элементов со значением 5
 vType (v,"After insert");

 p = v.begin();
 p += 2;
 v.erase (p, p + 10); // удалить вставленные элементы
 vType (v,"After delete");

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

В данном случае правильно определить тип операнда для оператора "=" в функции vType помог тип данных auto.

Вектор не обязан состоять из значений простого типа данных, в примере ниже вектор содержит объекты пользовательского класса Student.

Пример 5. Вектор из объектов пользовательского класса.

```
#include <iostream>
#include <vector>
using namespace std;

class Student {
public:
 char *Name;
 double value;
 Student (char *_Name=0,double _value=0.) : Name(_Name), value(_value) {}
 template <class OutStudent> friend ostream & operator <<
   (OutStudent &os, const Student &stud) {
  os << stud.Name << "," << stud.value;
  return os;
 }
};

template <typename T> void vType (T v, char *msg) {
 cout << endl << msg << ": ";
 for (auto p=v.begin(); p<v.end(); p++) cout << *p << " ";
}

int main(void) {
```

```
 vector <Student> v(3);
 v[0] = Student("Ivanoff", 45.9);
 v[1] = Student("Petroff", 30.4);
 v[2] = Student("Sidoroff", 55.6);
 vType(v,"List");
 cin.get(); return 0;
}
```

Для того, чтобы в отношении объектов класса Student работал вывод в cout оператором <<, понадобилось довольно нетривиальное решение для этого оператора. Без этого мы рисковали получить при компиляции ошибку "error C2679: бинарный " <<": не найден оператор, принимающий правый операнд типа "Student" (или приемлемое преобразование отсутствует)".

Так как встроенный в cout переопределённый оператор << умеет принимать объекты встроенных типов, более простым, но менее гибким решением было бы

```
class Student {
public:
 char *Name;
 double value;
 Student (char *_Name=0,double _value=0.) : Name(_Name), value(_value) {}
};

template <typename T> void vType (T v, char *msg) {
 cout << endl << msg << ": ";
 int n = v.size();
 for (int i=0; i<n; i++) cout <<
   v[i].Name << "," << v[i].value << " ";
}
```

В общем случае, конечно, ваши классы и шаблоны не будут "привязаны" к необходимости что-то выводить в консоль.

Теперь поговорим об *ассоциативных контейнерах*. В современных языках программирования ассоциативный массив содержит пары элементов "ключ"-"значение". Фактически, ассоциативный массив является расширением "обычного" массива C++, для которого ключами служат целые значения индекса 0,1,2 и так далее.

Зная ключ (key), мы можем получить доступ к отображаемому значению элемента ассоциативного массива (mapped value). В классе ассоциативного массива переопределённый оператор V &operator [](const K &), как правило, возвращает ссылку на значение V, соответствующее ключу K.

Ассоциативные контейнеры, в свою очередь, являются обобщением понятия ассоциативного массива.

Ассоциативный массив map - это последовательность пар {ключ, значение}, которая обеспечивает быстрое получение значения по ключу. Контейнер map предоставляет двунаправленные итераторы.

Контейнер map требует, чтобы для типов ключа существовала операция <. Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса map выглядит следующим образом:

```
template <class Key, class T, class Comp = less<Key>,
         class Allocator = allocator <pair> >
class std::map;
В классе map определены следующие конструкторы:
explicit map(const Comp &c = Comp(), const Allocator &a = Allocator());
 //пустой контейнер
map(const map<Key, T, Comp, Allocator> &ob);
 //конструктор копирования
template<class InIter> map(InIter first, InIter last, const Comp &c =
Comp(),const Allocator &a = Allocator());
 //конструктор контейнера, содержащего диапазон элементов
```

Определена операция присваивания:

```
map &operator =(const map &);
```

Определены операции сравнения ==, <, <=, !=, >, >=.

В map хранятся пары ключ/значение в виде объектов типа pair, указанного при создании контейнера.

Создавать пары "ключ/значение" можно не только с помощью конструкторов класса pair, но и с помощью функции make_pair, которая создает объекты типа pair, используя типы данных в качестве параметров.

Типичная операция для ассоциативного контейнера - это ассоциативный поиск при помощи операции индексации []:

```
mapped_type &operator [](const key_type &K);
```

Множества set можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи.

```
template<classT, class Cmp = less<T>, class Allocator = allocator<T> >
class std::set
{
  //...
};
```

Множество, как и ассоциативный массив, требует, чтобы для типа T существовала операция "меньше" (<). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Приведём пример реализации ассоциативного массива из пар "целый ключ"-"строковое значение" и "символьный ключ" – "целое значение". Обратите внимание, что двух значений с одинаковым ключом в map быть не может, если это нужно, используйте multimap.

Пример 6. Работа с ассоциативным массивом map.

```
#include <iostream>
#include <string>
#include <map> //подключили библиотеку для работы с map
using namespace std;

int main() {
```

```
 //Сделаем и инициализируем map
 map <int,string> myFirstMap;
 myFirstMap.insert(pair<int, string>(17, "Cooka"));
 myFirstMap.insert(pair<int, string>(40, "Marcooka"));
 myFirstMap.insert(pair<int, string>(17, "Balyam")); //не вставится!
 myFirstMap.insert( pair<int, string>(20, "Barabuka"));
 for (auto it = myFirstMap.begin(); it != myFirstMap.end(); it++)
  cout << it->first << " : " << it->second << endl;

 //Ещё один map
 char c='a';
 map <char, int> mySecondMap;
 for (int i = 0; i < 5; ++i,++c)
  mySecondMap.insert ( pair<char, int>(c,i+1));
 mySecondMap.insert ( pair<char, int>('d',5) ); //не вставится!
 mySecondMap.insert(mySecondMap.begin(), make_pair('e',1));
  //так тоже не вставится
 mySecondMap.insert(mySecondMap.begin(), make_pair('z',0) ); //а так можно
 for (auto it = mySecondMap.begin(); it != mySecondMap.end(); it++)
  cout << (*it).first << " : " << (*it).second << endl;

 cin.get(); return 0;
}
```

Пример ниже показывает простое множество символов. Обратите внимание на способ вывода множества в консоль и на то, что одинаковые по коду символы были исключены при добавлении.

Пример 7. Работа с множеством set.

```
#include <iostream>
#include <string>
#include <set>  // заголовочный файл множеств и мультимножеств
#include <iterator>
using namespace std;

int main() {
 set <char> mySet;
 string str("Abracadabra");
 for (int i=0; i<str.length(); i++) mySet.insert(str[i]);
 copy(mySet.begin(), mySet.end(), ostream_iterator<char>(cout, " "));
 cin.get(); return 0;
}
```

Наконец, в STL реализованы готовые *алгоритмы* для работы с элементами контейнеров. Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, один и тот же алгоритм может работать с разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их

входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в заголовочном файле <algorithm>, приведём имена некоторых наиболее распространённых функций-алгоритмов.

Основные алгоритмы STL

Функция	Описание
Немодифицирующие операции	
for_each()	выполняет операции для каждого элемента последовательности
find()	находит первое вхождение значения в последовательность
find_if()	находит первое соответствие предикату в последовательности
count()	подсчитывает количество вхождений значения в последовательность
count_if()	подсчитывает количество выполнений предиката в последовательности
search()	находит первое вхождение последовательности как подпоследовательности
search_n()	находит n-ое вхождение значения в последовательность
Модифицирующие операции	
copy()	копирует последовательность, начиная с первого элемента
swap()	меняет местами два элемента
replace()	заменяет элементы с указанным значением
replace_if()	заменяет элементы при выполнении предиката
replace_copy()	копирует последовательность, заменяя элементы с указанным значением
replace_copy_if()	копирует последовательность, заменяя элементы при выполнении предиката
fill()	заменяет все элементы заданным значением
remove()	удаляет элементы с заданным значением
remove_if()	удаляет элементы при выполнении предиката
remove_copy()	копирует последовательность, удаляя элементы с указанным значением
remove_copy_if()	копирует последовательность, удаляя элементы при выполнении предиката
reverse()	меняет порядок следования элементов на обратный
random_shuffle()	перемещает элементы согласно случайному равномерному распределению ("тасует" последовательность)
transform()	выполняет заданную операцию над каждым элементом последовательности
unique()	удаляет равные соседние элементы
unique_copy()	копирует последовательность, удаляя равные соседние элементы

Сортировка	
sort()	сортирует последовательность
partial_sort()	сортирует часть последовательности
stable_sort()	сортирует последовательность, сохраняя порядок следования равных элементов
lower_bound()	находит первое вхождение значения в отсортированной последовательности
upper_bound()	находит первый элемент, больший чем заданное значение
binary_search()	определяет, есть ли данный элемент в отсортированной последовательности
merge()	сливает две отсортированные последовательности
Работа с множествами	
includes()	проверка на вхождение
set_union()	объединение множеств
set_intersection()	пересечение множеств
set_difference()	разность множеств
Минимумы и максимумы	
min()	меньшее из двух значений
max()	большее из двух значений
min_element()	наименьшее значение в последовательности
max_element()	наибольшее значение в последовательности
Перестановки	
next_permutation()	следующая перестановка в лексикографическом порядке
prev_permutation()	предыдущая перестановка в лексикографическом порядке

Рассмотрим основные возможности алгоритмов на примерах.

Пример 8. 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);
 cin.get(); 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 должны работать и в версиях постарше.

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

Здесь формируется оглавление серии лекций

теги: алгоритм c++ учебное

Здесь можно оставить комментарий, обязательны только выделенные цветом поля. Не пишите лишнего, и всё будет хорошо

Ваше имя:

Пароль (если желаете