

Лекция. Стандартная библиотека шаблонов (STL).

Содержание

Введение в STL.....	2
Структура стандартной библиотеки шаблонов (STL).....	3
Контейнеры.....	4
Последовательные контейнеры.....	5
Ассоциативные контейнеры.....	11
Адаптеры контейнеров.....	15
Псевдоконтейнеры.....	18
Алгоритмы.....	21
Методы.....	24
Итераторы.....	25
Возможные проблемы с STL.....	26

Введение в STL

Большинство компьютеров предназначено для обработки информации. В качестве данных могут выступать самые разные виды характеристик реального мира: это может быть досье на работников, информация об имеющихся на складе запасах, текстовый документ, результат научных экспериментов и т. д. Что бы данные собой ни представляли, хранятся и обрабатываются они примерно одинаковыми способами. В университетские учебные планы по специальности «Информатика» обычно входит курс «Структуры данных и алгоритмы». Термин структура данных говорит о том, как хранится информация в памяти компьютера, а алгоритм — как эта информация обрабатывается.

Классы C++ предоставляют прекрасный механизм для создания библиотеки параметризованных абстракций данных. В прошлом производители компиляторов и разные сторонние разработчики ПО предлагали на рынке библиотеки классов-контейнеров для хранения и обработки данных. Теперь же в стандарт C++ входит собственная встроенная библиотека классов-контейнеров. Она называется Стандартной библиотекой шаблонов (в дальнейшем мы будем употреблять сокращение STL (<http://www.rsdn.ru/article/cpp/stl.xml>)) и разработана Александром Степановым и Менг Ли из фирмы Hewlett Packard. STL — это часть Стандартной библиотеки классов C++, которая может использоваться для хранения и обработки данных.

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнерные классы, а также алгоритмы и итераторы, о которых будет рассказано в следующих разделах, называют стандартной библиотекой шаблонов (STL — Standard Template Library).

Использование контейнеров позволяет значительно повысить

- надежность программ,
- их переносимость,
- универсальность,
- а также уменьшить сроки их разработки.

Естественно, эти преимущества не даются даром: универсальность и безопасность использования контейнерных классов не могут не отражаться на быстродействии программы. Снижение быстродействия в зависимости от реализации компилятора может быть весьма значительным. Кроме того, для эффективного использования контейнеров требуется затратить усилия на вдумчивое освоение библиотеки.

В данной лекции описывается принцип построения STL и работа с ней. Тема большая и сложная, и мы не будем описывать все, что входит в библиотеку. Мы ограничимся здесь тем, что представим STL и приведем примеры наиболее часто используемых алгоритмов и контейнеров.

Структура стандартной библиотеки шаблонов (STL)

В библиотеке выделяют пять основных компонентов:

1. Контейнер (англ. container) — хранение набора объектов в памяти.
2. Итератор (англ. iterator) — обеспечение средств доступа к содержимому контейнера.
3. Алгоритм (англ. algorithm) — определение вычислительной процедуры.
4. Адаптер (англ. adaptor) — адаптация компонентов для обеспечения различного интерфейса.
5. Функциональный объект (англ. functor) — сокрытие функции в объекте для использования другими компонентами.

Разделение позволяет уменьшить количество компонентов. Например, вместо написания отдельной функции поиска элемента для каждого типа контейнера обеспечивается единственная версия, которая работает с каждым из них, пока соблюдаются основные требования.

Три наиболее важные компонента STL — это контейнеры, алгоритмы и итераторы.

Контейнер — это способ организации хранения данных. Вам, наверняка, уже встречались некоторые контейнеры, такие, как стек, связный список, очередь. Еще один контейнер — это массив, но он настолько тривиален и популярен, что встроен в C++ и большинство других языков программирования. Контейнеры бывают самые разнообразные, и в STL включены наиболее полезные из них. Контейнеры STL подключаются к программе с помощью шаблонов классов, а значит, можно легко изменить тип хранимых в них данных.

Под **алгоритмами** в STL понимают процедуры, применяемые к контейнерам для обработки содержащихся в них данных различными способами. Например, есть алгоритмы сортировки, копирования, поиска и объединения. Алгоритмы представлены в STL в виде шаблонов функций. Однако они не являются методами классов-контейнеров. Наоборот, это совершенно независимые функции. На самом деле, одной из самых привлекательных черт STL является универсальность ее алгоритмов. Их можно использовать не только для объектов классов-контейнеров, но и для обычных массивов и даже для собственных контейнеров. (Контейнеры, тем не менее, содержат методы для выполнения некоторых специфических задач.)

Итераторы — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, и они будут ссылаться последовательно на все элементы контейнера. Итераторы — ключевая часть всего STL, поскольку они связывают алгоритмы с контейнерами. Их можно представить себе в виде кабеля, связывающего колонки вашей стереосистемы или компьютер с его периферией.

На рис. 1 показаны три компонента STL. В этом разделе мы обсудим их более детально.

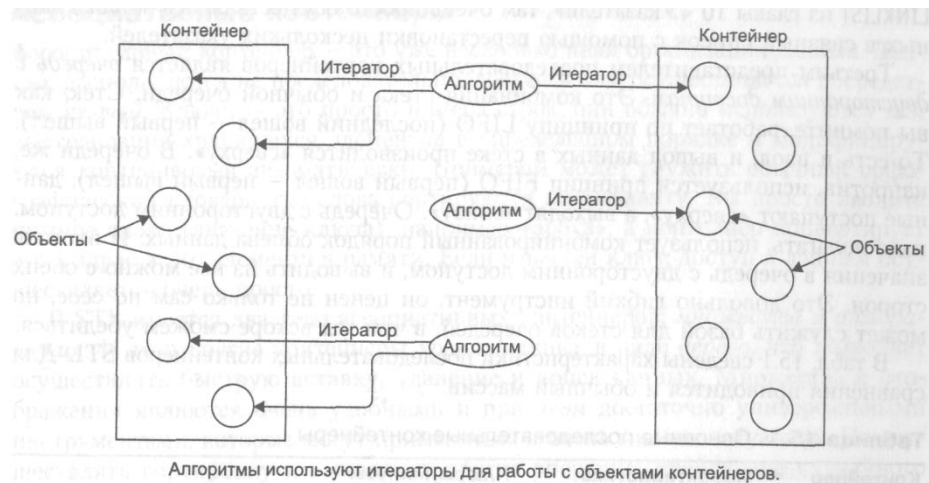


Рис. 1. Контейнеры, алгоритмы и итераторы.

Компоненты STL определяются в 14 заголовочных файлах:

algorithm	array	deque	functional	iterator	list	map
memory	numeric	queue	set	stack	utility	vector

Контейнеры

Контейнерные классы обеспечивают стандартизованный интерфейс при их использовании. Смысл одноименных операций для различных контейнеров одинаков, основные операции применимы ко всем типам контейнеров. Стандарт определяет только интерфейс контейнеров, поэтому разные реализации могут сильно отличаться по эффективности.

Как мы уже говорили, контейнеры представляют собой различные параметризованные абстракции данных, объекты которых предназначены для хранения и обработки данных различных типов. При этом не имеет значения, какие именно данные хранятся, будь то некоторые базовые типы, такие, как `int`, `float` и т. п., или же объекты классов. **STL включает в себя восемь основных типов контейнеров и еще три производных типа.** Зачем же нам столько типов контейнеров? Почему нельзя использовать массив во всех случаях, когда нужно хранить данные? Ответ таков: неэффективно. Работать с массивом во многих ситуациях бывает неудобно — медленно и вообще затруднительно.

Контейнеры библиотеки STL можно разделить на четыре категории:

- последовательные,
- ассоциативные,
- контейнеры-адаптеры,
- псевдоконтейнеры.

Среди последовательных выделяют:

- массивы (`array`),
- списки,
- векторы,
- очереди с двусторонним доступом.

Среди ассоциативных —

- множества,
- мультимножества,

- отображения,
- мультиотображения.

Кроме того, наследниками последовательных выступают еще несколько контейнеров-адаптеров:

- стек,
- очередь,
- очередь приоритетов.

Рассмотрим пункты этой классификации более подробно.

Последовательные контейнеры

В последовательных контейнерах данные хранятся подобно тому, как дома стоят на улице — в ряд. Каждый элемент связывается с другими посредством номера своей позиции в ряду. Все элементы, кроме крайних, имеют по одному соседу с каждой стороны. Примером последовательного контейнера является обычный массив.

Проблема с массивами заключается в том, что их размеры нужно указывать при компиляции, то есть в исходном коде. К сожалению, во время написания программы обычно бывает неизвестно, сколько элементов потребуется записать в массив. Поэтому, как правило, рассчитывают на худший вариант, и размер массива указывают «с запасом». В итоге при работе программы выясняется, что либо в массиве остается очень много свободного места, либо все-таки не хватает ячеек, а это может привести к чему угодно, вплоть до аварийного завершения программы. Для борьбы с такой проблемой в STL существует контейнер (*vector*).

Кроме того, при передаче массива в функцию в качестве параметра, массив распадается на отдельные указатели, поскольку передаётся указатель на первый элемент массива. Поэтому вместе с указателем на первый элемент массива приходится передавать и его размер. Эту проблему решает последовательный контейнер (*array*), который представляет собой аналог массива.

Еще одна проблема существует при работе с массивами. Пускай, в массиве хранятся данные о работниках, отсортированные по алфавиту в соответствии с фамилиями. Если необходимо добавить сотрудника, фамилия которого начинается с буквы Л, то придется сдвинуть всех работников с фамилиями, начинающимися с букв от М до Я, чтобы освободить место для вставки нового значения. Сдвиг данных может занимать довольно много времени. Контейнер STL список (*list*), который основан на идее связного списка, решает данный вопрос.

Четвёртым представителем последовательных контейнеров является очередь с двусторонним доступом (*deque*). Это комбинация стека и обычной очереди. Стек, как вы помните, работает по принципу LIFO (последний вошел — первый вышел). То есть и ввод, и вывод данных в стеке производится «через вершину стека». В очереди же, напротив, используется принцип FIFO (первый вошел — первый вышел): данные поступают «в начало очереди», а выходят «из конца очереди». Очередь с двусторонним доступом, использует комбинированный порядок обработки данных. В очередь с двусторонним доступом как вносить значения, так и выводить их из нее можно с обеих сторон. Это довольно гибкий инструмент, он ценен не только сам по себе, но на его основе можно организовать стеки и очереди.

В табл.1 сведены характеристики последовательных контейнеров STL. Для сравнения приводится и обычный массив.

Таблица 1. Основные последовательные контейнеры.

Контейнер	Характеристика	Плюсы/минусы
Обычный массив	Фиксированный размер	<p>Скоростной случайный доступ (по индексу)</p> <p>Медленная вставка или изъятие данных из середины</p> <p>Размер не может быть изменен во время работы программы</p>
Вектор	Перераспределяемый, расширяемый массив	<p>Скоростной случайный доступ (по индексу)</p> <p>Медленная вставка или изъятие данных из середины</p> <p>Скоростная вставка или изъятие данных из хвоста</p>
Список	Аналогичен связанному списку	<p>Скоростная вставка или изъятие данных из любого места</p> <p>Быстрый доступ к обоим концам</p> <p>Медленный случайный доступ</p>
Очередь с двусторонним доступом	Как вектор, но доступ с обоих концов	<p>Скоростной случайный доступ</p> <p>Медленная вставка или изъятие данных из середины</p> <p>Скоростная вставка и изъятие данных из хвоста или головы</p>

Использование контейнера STL не представляет особого труда. Во-первых, необходимо включить в программу соответствующий заголовочный файл. Передача информации о том, какие типы объектов будут храниться, производится посредством передачи параметра в шаблон.

Например, в следующей программе создаётся массив (`array`) из пяти целых чисел (типа `int`):

```

// Пример 1. Array.cpp: определяет точку входа для консольного
приложения.
#include "stdafx.h"
#include <iostream>
#include <array>

using namespace std;

typedef array<int, 5> MyArray; //Описание нового типа.

//Шаблон функции для вывода элементов контейнера на консоль.
template <class T>
void Print(const T& myarr)
{
    for (const auto & x : myarr)
        cout << x << ' ';
    cout << endl;
}

int main()
{
    array<int, 5> myarr; //Объявляем массив типа int длиной 5.
    myarr = { 2, 4, 5, 6, 8 }; //Инициализируем массив начальными
значениями.
    Print(myarr);
    array<int, 5> myarr1 = { 1, 5, 3, 7, 9 }; //Объявляем и
инициализируем.

    array<int, 5> myarr2 { 5, 4, 3, 2, 1 }; //Объявляем и
инициализируем.

    MyArray myarr3(myarr); //Объявляем и инициализируем.
    myarr3 = { 7, 9 }; //Заменяем первые два значения, остальные
обнуляем.
    Print(myarr3);

    return 0;
}

```

```

2 4 5 6 8
7 9 0 0 0
Для продолжения нажмите любую клавишу . . . _

```

Например, в следующей программе создаётся вектор целых чисел (типа `int`):

```
//-----
```

```

// Пример 2. Вектор.cpp: определяет точку входа для консольного
приложения.
//
#include "stdafx.h"
#include <iostream>
#include <vector>
#include "windows.h"

using namespace std;

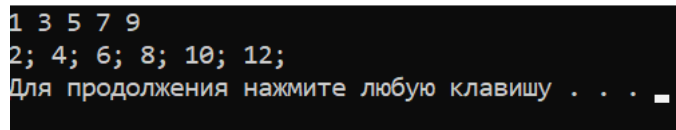
typedef vector<int> MyVector; //Описание нового типа.

template <class T> //Шаблон функции для вывода элементов контейнера на
консоль.
void Print(const T& myarr)
{
    for (const auto & x : myarr)
        cout << x << ' ';
    cout << endl;
}

int main()
{
    int m[5] = { 1, 3, 5, 7, 9 };
    vector<int> a(m, m + 5); //Вектор инициализируется значениями
массива m.
    Print(a); //Просмотреть все компоненты вектора.
    MyVector b{ 2, 4, 6, 8, 10 };
    MyVector::iterator iter; //Итератор для вектора.
    b.push_back(12); //Добавить в конец вектора.
    for (iter = b.begin(); iter != b.end(); iter++) //Просмотреть все
компоненты вектора.
        cout << *iter << " ";
    cout << endl;
    system("PAUSE");
    return 0;
} //-----
-----

```

Результат работы программы:



```

1 3 5 7 9
2; 4; 6; 8; 10; 12;
Для продолжения нажмите любую клавишу . . . _

```

или

```
//создать список типа airtime
```

```
list<airtime> d;
```

В следующем ниже примере создаётся список целых чисел.

```
//-----
```



```

// Пример 3. Список.cpp: определяет точку входа для консольного
приложения.
//
#include "stdafx.h"
#include <iostream>
#include <list>
#include <array>
#include "windows.h"

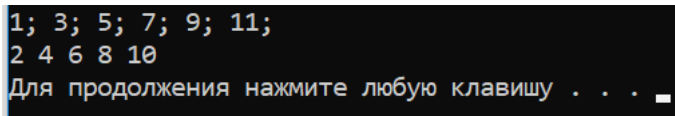
using namespace std;

//Шаблон функции для вывода элементов контейнера на консоль.
template <class T>
void Print(const T& myarr)
{
    for (const auto & x : myarr)
        cout << x << ' ';
    cout << endl;
}

int main()
{
    int m[5] = { 1, 3, 5, 7, 9 };
    list<int> a(m, m + 5); //Список инициализируется значениями
массива m.
    list<int> :: iterator iter; //Итератор для списка.
    a.push_back(11); //Добавить в конец списка.
    for (iter = a.begin(); iter != a.end(); iter++) //Просмотреть все
компоненты списка.
        cout << *iter << "; ";
    cout << endl;
    array<int, 5> b { 2, 4, 6, 8, 10 };
    list<int> c(b.begin(), b.end()); //Список инициализируется
значениями массива m.
    Print(c); //Просмотреть все компоненты списка.
    system("PAUSE");
    return 0;
}

```

Результат работы программы:



```

1; 3; 5; 7; 9; 11;
2 4 6 8 10
Для продолжения нажмите любую клавишу . . .

```

Обратите внимание: в STL не нужно специфицировать размеры контейнеров. Они сами заботятся о размещении своих данных в памяти.

В примере, следующем ниже, создаётся очередь целых чисел (дека) с двухсторонним доступом.

```

// Пример 4. Дека.cpp: определяет точку входа для консольного
приложения.

#include "stdafx.h"
#include <iostream>
#include <deque>
#include "windows.h"

using namespace std;

template <class T> //Шаблон функции для вывода элементов контейнера на
консоль.
void Print(const T& myarr)
{
    for (const auto & x : myarr)
        cout << x << ' ';
    cout << endl;
}

int main()
{
    int m[5] = { 1,3,5,7,9 };
    deque<int> a(m, m + 5); //Дека инициализируется значениями массива
m.
    Print(a); //Просмотреть все компоненты дека.
    deque<int>::iterator iter; // Итератор дека.
    a.push_front(0); //Добавить в начало дека.
    a.push_back(11); //Добавить в конец дека.
    for (iter = a.begin(); iter != a.end(); iter++) //просмотреть
компоненты дека
        cout << *iter << "; ";
    cout << endl;
    cout << a.front() << "; " << a.back() << endl; //Просмотреть
крайние компоненты.
    a.pop_back(); a.pop_front(); //Удалить крайние компоненты.
    for (int i = 0; i < a.size(); i++) //Просмотреть компоненты дека.
        cout << a[i] << "; ";
    cout << endl;
    system("PAUSE");
    return 0;
}

```

//-----

Результат работы:

```

1 3 5 7 9
0; 1; 3; 5; 7; 9; 11;
0; 11
1; 3; 5; 7; 9;
Для продолжения нажмите любую клавишу . . .

```

Ассоциативные контейнеры

Ассоциативный контейнер — это уже несколько иная организация данных. Данные располагаются не последовательно, доступ к ним осуществляется посредством ключей. Ключи обычно используются для выстраивания хранящихся элементов в определенном порядке. Примером может служить обычный орфографический словарь, где слова сортируются по алфавиту. Вы просто вводите нужное слово (значение ключа), например «арбуз», а контейнер конвертирует его в адрес этого элемента в памяти. Если известен ключ, доступ к данным осуществляется очень просто.

В STL имеется два типа ассоциативных контейнеров: множества (set) и отображения (map). И те и другие контейнеры хранят данные в виде дерева, что позволяет осуществлять быструю вставку, удаление и поиск данных. Множества и отображения являются очень удобными и при этом достаточно универсальными инструментами, которые могут применяться в очень многих ситуациях. Но осуществлять сортировку и выполнять другие операции, требующие случайного доступа к данным, неудобно.

Множества проще и, возможно, из-за этого более популярны, чем отображения. Множество хранит набор элементов, содержащих ключи — атрибуты, используемые для упорядочивания данных. Например, множество может хранить объекты класса person, которые упорядочиваются в алфавитном порядке.

```

class person
{
private:
    string lastName;
    string firstName;
    long phoneNumber;
public:
    // Конструктор по умолчанию.
    person(): lastName("пусто"), firstName("пусто"), phoneNumber(0)
    { }
    // Конструктор с тремя параметрами.
    person(string lana, string fina, long pho):
        lastName(lana), firstName(fina), phoneNumber(pho)
    { }
    friend bool operator<(const person&, const person&);
    friend bool operator==(const person&, const person&);
    // Строковое представление объекта.
    string ToString (void)
    {
        ostringstream os;
        os << endl << lastName << ",\t" << firstName << "\t\Номер телефона: " << phoneNumber;
        return os.str();
    }
};
// Оператор < для класса person.
bool operator<(const person& p1, const person& p2)
{
    if(p1.lastName == p2.lastName)
        return (p1.firstName < p2.firstName) ? true : false;
    return (p1.lastName < p2.lastName) ? true : false;
}
// Оператор == для класса person.
bool operator==(const person& p1, const person& p2)
{
    return (p1.lastName == p2.lastName && p1.firstName == p2.firstName) ? true : false;
}

```

В качестве ключа при этом используется значение имя работника (поля lastName, firstName). При такой организации данных можно очень быстро найти нужный объект, осуществляя поиск по имени объекта (ищем объект класса person по атрибутам lastName, firstName). Если во множестве хранятся значения одного из базовых типов, таких, как int, ключом является сам элемент. Некоторые авторы называют ключом весь объект, хранящийся во множестве, но мы будем употреблять термин ключевой объект, чтобы подчеркнуть, что атрибут, используемый для упорядочивания (ключ), — это не обязательно весь элемент данных.

Отображения (map) часто называют словарями или ассоциативными массивами. Отображение хранит пары объектов. Один кортеж в такой «базе данных» состоит из двух «атрибутов»: **ключевой объект и целевой (ассоциированный) объект**. Этот инструмент часто используется в качестве контейнера, несколько напоминающего массив. Разница лишь в том, что доступ к данным осуществляется не по номеру элемента, а по специальному индексу, который сам по себе может быть произвольного типа. **То есть ключевой объект служит индексом, а целевой — значением этого индекса.**

Контейнеры отображение и множество разрешают сопоставлять только один ключ данному значению. Это имеет смысл в таких, например, случаях, как хранение списка работников, где каждому имени сопоставляется уникальный идентификационный номер. **С другой стороны, мультиотображения (multimap) и мультимножества (multiset) позволяют хранить несколько одинаковых ключей.** Например, слову «ключ» в толковом словаре может быть сопоставлено несколько статей. Поэтому для них не определена операция доступа по индексу.

В табл. 15.2 собраны все ассоциативные контейнеры, имеющиеся в STL.

Контейнер	Характеристики
Множество	Хранит только ключевые объекты. Каждому ключу сопоставлено одно значение
Мультимножество	Хранит только ключевые объекты. Каждому ключу может быть сопоставлено несколько значений
Отображение	Ассоциирует ключевой объект с объектом, хранящим значение (целевым). Одному ключу сопоставлено одно значение.
Мультиотображение	Ассоциирует ключевой объект с объектом, хранящим значение (целевым). Одному ключу может быть сопоставлено несколько значений.

Создаются ассоциативные контейнеры примерно так же, как и последовательные:

```
set<int> intSet; //создает множество значений int.
```

В следующем ниже примере создаётся множество целых чисел.

// Пример 5. Множество.cpp: определяет точку входа для консольного приложения.

```
#include "stdafx.h"
#include <iostream>
#include <deque>
#include <set>
#include "windows.h"
```

```
using namespace std;
```

```
template <class T> //Шаблон функции вывода элементов контейнера на консоль.
```

```
void Print(const T& myarr)
{
    for (const auto & x : myarr)
        cout << x << ' ';
    cout << endl;
}
```

```
int main()
{
```

```
    deque<int> a{ 1,3,5,7,9 }; //Дека.
    Print(a); //Просмотреть все компоненты дека.
    //Множество целых упорядоченное с помощью greater.
    set<int, greater<int> > s(a.begin(), a.begin() + 3);
```

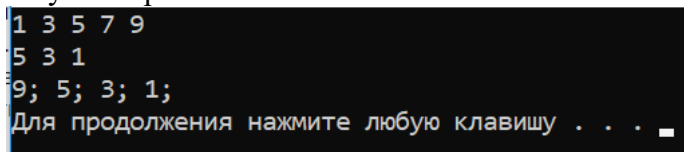
```

    // greater<int> - функциональный объект, используемый для
упорядочения в порядке убывания.
    Print(s);
    set<int, greater<int> >::iterator iter;//Итератор для множества.

    s.insert(9); //Вставить 9.
    //Выводим компоненты множества на консоль.
    for (iter = s.begin(); iter != s.end(); iter++)
        cout << *iter << "; ";
    cout << endl;
    system("PAUSE");
    return 0;
}

```

Результат работы:



```

1 3 5 7 9
5 3 1
9; 5; 3; 1;
Для продолжения нажмите любую клавишу . . .

```

В следующем примере создаётся отображение:

// Пример 6.Отображение.cpp: определяет точку входа для консольного приложения.

```

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

using namespace std;

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    //Имя города.
    string name;
    //Население города в тысячах человек.
    int pop;
    string townes[] = { "Екатирибург", "Нижий Новгород",
"Новосибирск",
    "Омск", "Ростов-на-Дону", "Уфа", "Челябинск" };
    int pops[] = { 1350, 1300, 1500, 1150, 1050, 1100, 1130 };

    //Отображение строки в целое.
    map<string, int, less<string> > mapTownes;
    //less<string> - функциональный объект, используемый для
упорядочения в порядке возрастания.
    //Итератор для доступа к объектам.
    map<string, int>::iterator iter;

```

```

//Используется по умолчанию less<string>.
for (int j = 0; j < 6; j++)
{
    name = townes[j];
    pop = pops[j];
    mapTownes[name] = pop;
}
//Вывод на консоль ключевого и целевого объектов отображения.
for (const auto & x : mapTownes)
    cout << x.first << ' ' << x.second << endl;
cout << endl;
cout << "Имя города: ";
cin >> name;
pop = mapTownes[name];
cout << "Population: " << pop << endl;
//Вывод на консоль ключевого и целевого объектов отображения.
for (iter = mapTownes.begin(); iter != mapTownes.end(); iter++)
    cout << iter->first << ' ' << iter->second << "000\n";
cout << endl;

mapTownes["Омск"] = 1160;
cout << mapTownes["Омск"] << endl;
system("PAUSE");
return 0;
}

```

```

//-----
Екатирибург 1350
Нижий Новгород 1300
Новосибирск 1500
Омск 1150
Ростов-на-Дону 1050
Уфа 1100

Имя города: Уфа
Population: 1100
Екатирибург 1350000
Нижий Новгород 1300000
Новосибирск 1500000
Омск 1150000
Ростов-на-Дону 1050000
Уфа 1100000

1160
Для продолжения нажмите любую клавишу . . .

```

Адаптеры контейнеров

Специализированные контейнеры можно создавать из базовых (приведенных выше) с помощью конструкции, называющейся адаптером контейнера. Они обладают более простым интерфейсом, чем обычные контейнеры. Специализированные контейнеры, реализованные в STL, это стеки (stack), очереди (queue) и очереди приоритетов (priority_queue). Как уже отмечалось, для стека характерен доступ к данным только с одного конца, его можно сравнить со стопкой книг. Очередь использует для проталкивания данных один конец, а для выталкивания — другой. В очереди приоритетов

данные проталкиваются спереди в произвольном порядке, а выталкиваются в строгом соответствии с величиной приоритета хранящегося значения. В простейшем случае, приоритет имеют данные с наибольшим значением. Таким образом, очередь приоритетов автоматически сортирует хранящуюся в ней информацию.

Стеки, очереди и очереди приоритетов – адаптеры, которые могут создаваться из разных последовательных контейнеров. Хотя очередь с двусторонним доступом позволяет создать любой из них. В табл. 4 показаны адаптеры и последовательные контейнеры, использующиеся для их реализации.

Таблица 4. Адаптеры, реализуемые с помощью контейнеров.

Адаптер	Контейнер для реализации	Характеристики
Стек	Реализуется на векторе, списке или очереди с двухсторонним доступом	Проталкивание и выталкивание данных с одного конца
Очередь	Реализуется на списке или очереди с двухсторонним доступом	Проталкивание данных с одного конца, выталкивание – с другого
Очередь приоритетов	Реализуется на векторе или очереди с двухсторонним доступом	Проталкивание данных с одного конца случайным образом, выталкивание – с другого конца в соответствии с порядком

Для практического применения этих классов необходимо использовать как бы шаблон в шаблоне. Например, пусть имеется объект типа стек, содержащий значения `int`, порожденный классом «очередь с двусторонним доступом» (`deque`):

```
stack< int, deque<int> > aStak;
```

Деталь, на которую стоит обратить внимание при описании этого формата, это пробелы, которые необходимо ставить между двумя закрывающими угловыми скобками. Помните об этом, потому что такое выражение `stack<deque<int> aStak;` приведет к синтаксической ошибке — компилятор интерпретирует »как оператор. Объекты очереди и очереди приоритетов будут иметь следующий вид:

```
queue <int, deque <int> > q;//очередь
```

```
priority_queue <int, vector <int>, less<int> > p;//очередь  
приоритетов
```

Псевдоконтейнеры

Псевдоконтейнеры	
bitset	Служит для хранения битовых масок. Похож на <code>vector<bool></code> фиксированного размера. Размер фиксируется тогда, когда объявляется объект <code>bitset</code> . Итераторов в <code>bitset</code> нет. Оптимизирован по размеру памяти.
valarray	Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности. В некоторой степени похож на <code>vector</code> , но в нём отсутствует большинство стандартных для контейнеров операций. Определены операции над двумя <code>valarray</code> и над <code>valarray</code> и скаляром (поэлементные). Эти операции возможно эффективно реализовать как на векторных процессорах, так и на скалярных процессорах с блоками SIMD.
basic_string	<p>Контейнер, предназначенный для хранения и обработки строк. Хранит в памяти элементы подряд единым блоком, что позволяет организовать быстрый доступ ко всей последовательности. Определена конкатенация с помощью <code>+</code>.</p> <pre>typedef basic_string<char, char_traits<char>, allocator<char>> string;</pre> <p>Следующие объявления являются равнозначными:</p> <pre>string str(""); basic_string<char> str("");</pre>

Пример работы с псевдо контейнером `bitset`:

```
// Листинг 7.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <bitset>
#include "windows.h"

////////////////////////////////////
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
```

```

// Используем конструктор по умолчанию.
bitset<2> b0;
cout << "Последовательность бит в bitset<2> b0 : ( " << b0 << " )."
<< endl;

// Используем конструктор с параметром беззнаковое длинное целое.
bitset<5> b1( 6 );
cout << "Последовательность бит в bitset<5> b1( 6 ): ( " << b1 << "
)." << endl;
cout << "Последовательность бит в bitset<5> ~b1( 6 ): ( " << ~b1 <<
" )." << endl;
b1[0] = 1; //устанавливаем нулевой бит в единицу
cout << "Последовательность бит в bitset<5> b1( 6 ): ( " << ~b1 <<
" )." << endl;
system("PAUSE");
return 0;
}

```

Результат работы программы:

```

Последовательность бит в bitset<2> b0 : ( 00 ).
Последовательность бит в bitset<5> b1( 6 ): ( 00110 ).
Последовательность бит в bitset<5> ~b1( 6 ): ( 11001 ).
Последовательность бит в bitset<5> b1( 6 ): ( 11000 ).
Для продолжения нажмите любую клавишу . . .

```

Пример работы с псевдо контейнером valarray:

// Листинг 8.cpp: определяет точку входа для консольного приложения.
 //

```

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

using namespace std;

template <class T> //Шаблон функции вывода элементов контейнера на
консоль.
void Print(const T& cntnr)
{
    for (const auto & x : cntnr)
        cout << x << ' ';
    cout << endl;
}

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int i, MaxValue;
    int m[5] = { 3,6,4,8,9 };
}

```

```

    valarray<int> v(m, 5), v1(5), v2(5);
    //Получаем массив v1 циклическим сдвигом на одну позицию влево
    массива v.
    v1 = v.cshift(1);
    cout << "Массив v: ( ";
    for (i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << ")." << endl;

    cout << "Массив v1: ";
    Print(v1);
    //Получаем массив v2 сложением массива v с v1.
    v2 = v1 + v;
    cout << "Массив v1: ( ";
    for (i = 0; i < v2.size(); i++)
        cout << v2[i] << " ";
    cout << ")." << endl;
    system("PAUSE");
    return 0;
}

```

Результат работы программы:

```

Массив v: ( 3 6 4 8 9 ).
Массив v1: 6 4 8 9 3
Массив v1: ( 9 10 12 17 12 ).
Для продолжения нажмите любую клавишу . . .

```

Пример работы с псевдоконтейнером string/

Последовательности, управляемые объектом класса шаблона `basic_string`, являются строковым классом стандарта C++ и обычно называются просто строками, но их не следует путать с C-строками, оканчивающимися нулевым символом, в библиотеке Standard C++. Стандартная строка C++ является контейнером, позволяющим использовать строки как обычные типы, например, в операциях сравнения и сцепления, итераторах, алгоритмах STL, а также копировать и назначать с помощью управляемой памяти класса распределителя. Если необходимо преобразовать стандартную строку C++ в C-строку, оканчивающуюся нулевым символом, используйте член `basic_string::c_str`.

// Листинг 9.cpp: определяет точку входа для консольного приложения.
//

```

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

using namespace std;

template <class T> //Шаблон функции вывода элементов контейнера на
консоль.
void Print(const T& cntnr)
{
    for (const auto & x : cntnr)
        cout << x << ' ';
    cout << endl;
}

```

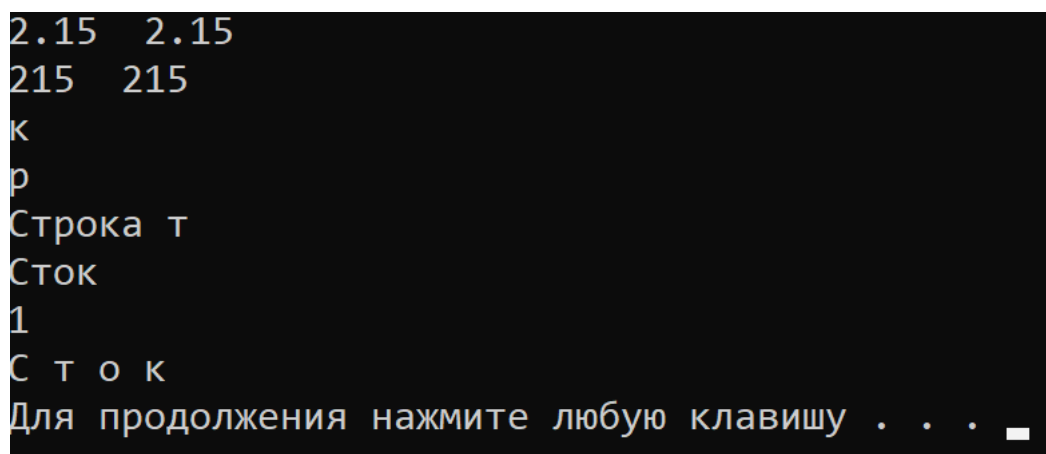
```

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    string s = "Строка";
    string sd("2.15");
    //Специализированная шаблонная функция.
    double d = stod(sd);
    cout << sd << " " << d << endl;
    string si("215");
    //Специализированная шаблонная функция.
    int n = stoi(si);
    cout << si << " " << n << endl;
    //Итератор для строк.
    string::iterator it = s.begin();
    it += 4; //Пятая позиция в строке.
    cout << *it << endl; //Символ к.
    it -= 2; //Третья позиция в строке.
    cout << *it << endl; //Символ р.
    cout << s << " " << s[1] << endl; //Строка т
    s.erase(2, 1);
    s.erase(4, 1);
    cout << s << endl; // Сток
    bool b = s.at(1) == s[1];
    cout << b << endl; // 1
    Print(s);
    return 0;
}

```

Результат работы программы:



```

2.15 2.15
215 215
к
р
Строка т
Сток
1
С т о к
Для продолжения нажмите любую клавишу . . .

```

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). В стандарте языка алгоритмы — это независимые шаблоны функций. Их можно использовать при работе как с обычными массивами C++, так и с вашими собственными классами-контейнерами (предполагается, что в класс включены

уже некоторые базовые функции).

В табл. 6 показаны некоторые часто используемые алгоритмы.

Таблица 6. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
find	Возвращает первый элемент с указанным значением
count	Считает количество элементов, имеющих указанное значение
equal	Сравнивает содержимое двух контейнеров и возвращает true, если все соответствующие элементы эквивалентны
search	Ищет последовательность значений в одном контейнере, которая соответствует такой же последовательности в другом
copy	Копирует последовательность значений из одного контейнера в другой (или в другое место того же контейнера)
swap	Обменивает значения, хранящиеся в разных местах
iter_swap	Обменивает последовательности значений, хранящиеся в разных местах
fill	Копирует значение в последовательность ячеек
sort	Сортирует значения в указанном порядке
merge	Комбинирует два сортированных диапазона значений для получения наибольшего диапазона
accumulate	Возвращает сумму элементов в заданном диапазоне
for_each	Выполняет указанную функцию для каждого элемента контейнера

Разработчики библиотеки STL ставили перед собой гораздо более серьезную задачу, чем создание библиотеки с набором шаблонных структур данных. STL содержит огромный набор оптимальных реализаций популярных алгоритмов, позволяющих работать с STL-коллекциями. Все реализованные функции можно поделить на три группы:

- Методы перебора всех элементов коллекции и их обработки: count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search, copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition
- Методы сортировки коллекции: sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexicographical_compare, next_permutation, prev_permutation
- Методы выполнения определенных арифметических операций над членами коллекций: Accumulate, inner_product, partial_sum, adjacent_difference

Предикаты

Для многих алгоритмов STL можно задать условие, посредством которого алгоритм определит, что ему делать с тем или иным членом коллекции. Предикат — это функция, которая принимает несколько параметров и возвращает логическое значение (истина/ложь). Существует и набор стандартных предикатов.

Допустим, вы создаете массив типа int со следующими данными:

```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```

Применим алгоритм sort() для сортировки массива:

sort(arr, arr+8);

где arr — это адрес начала массива, arr+8 — адрес конца (элемент, располагающийся позади последнего в массиве).

```

//-----
// Пример 9. Сортировка массива.cpp: определяет точку входа для
консольного приложения.
//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <algorithm>

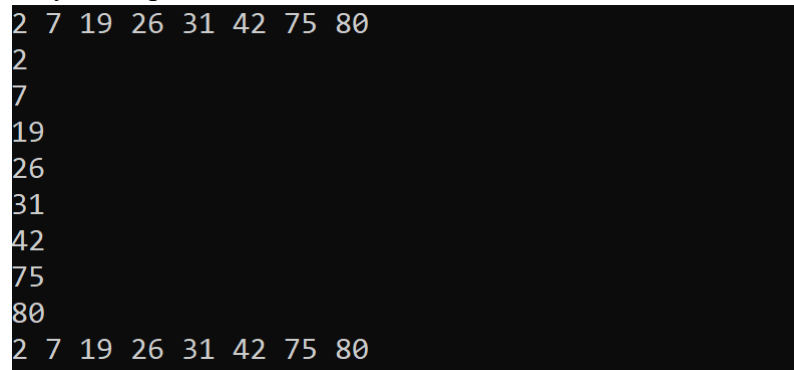
using namespace std;

template <class T> //Шаблон функции вывода элементов контейнера на
консоль.
void Print(const T& cntnr)
{
    for (const auto & x : cntnr)
        cout << x << ' ';
    cout << endl;
}
void Show(int x)
{
    cout << x << endl;
}

int main()
{
    int m[8] = { 42, 31, 7, 80, 2, 26, 19, 75 };
    sort(m, m + 8);
    Print(m);
    for_each(m, m + 8, Show);
    Print(m);
    system("pause");
    return 0;
}

```

Результат работы:



```

2 7 19 26 31 42 75 80
2
7
19
26
31
42
75
80
2 7 19 26 31 42 75 80

```

Методы

Алгоритмы используют для выполнения сложных операций типа сортировки и поиска. Для выполнения более простых операций, специфичных для конкретного контейнера, используют методы.

В табл. 5 представлены некоторые наиболее часто используемые методы, имена и назначение которых общие для большинства классов контейнеров.

Таблица 5. Некоторые методы, общие для всех контейнеров

Имя	Назначение
size()	Возвращает число элементов в контейнере
empty()	Возвращает true, если контейнер пуст
max_size()	Возвращает максимально допустимый размер контейнера
begin()	Возвращает итератор на начало контейнера (итерации будут проводиться в прямом направлении)
end()	Возвращает итератор на последнюю позицию контейнера (итерации в прямом направлении будут закончены)
rbegin()	Возвращает реверсивный итератор на конец контейнера (итерации будут проводиться в обратном направлении)
rend()	Возвращает реверсивный итератор на начало контейнера (итерации в обратном направлении будут завершены)

Существует еще множество методов, которые применяются в конкретных контейнерах или категориях контейнеров.

Итераторы

Итераторы — это сущности, напоминающие указатели. Они используются для получения доступа к отдельным данным (которые обычно называются элементами) в контейнере. Они часто используются для последовательного продвижения по контейнеру от элемента к элементу (этот процесс называется итерацией). Итераторы можно инкрементировать с помощью обычного оператора ++, после выполнения которого, итератор передвинется на следующий элемент. Косвенность со ссылок снимается оператором *, после чего можно получить значение элемента, на который ссылается итератор. В STL итератор представляет собой объект класса iterator.

Для разных типов контейнеров используются свои итераторы. Всего существует три основных класса итераторов: *прямые*, *двунаправленные* и *со случайным доступом*. Прямой итератор может проходить по контейнеру только в прямом направлении, что и указано в его названии. Проход осуществляется поэлементный. Работать с ним можно, используя ++. Такой итератор не может двигаться в обратном направлении и не может быть поставлен в произвольное место контейнера. Двунаправленный итератор, соответственно, может передвигаться в обоих направлениях и реагирует как на ++, так и на --. Итератор со случайным доступом может и двигаться в обоих направлениях, и перескакивать на произвольное место контейнера. Можно приказывать ему получить доступ к позиции 27, например.

Есть два специализированных вида итераторов. Это входной итератор, который может «указывать» на устройство ввода (cin или даже просто входной файл) и считывать последовательно элементы данных в контейнер, и выходной итератор, который, соответственно, указывает на устройство вывода (cout) или выходной файл и выводит элементы из контейнера.

В то время как значения прямых, двунаправленных итераторов и итераторов со случайным доступом могут быть сохранены, значения входных и выходных итераторов сохраняться не могут. Это имеет смысл, ибо первые три итератора все-таки указывают на некоторый адрес в памяти, тогда как входные и выходные итераторы указывают на устройства ввода/вывода, для которых хранить какой-то «указатель» невозможно. В табл. 7 показаны характеристики различных типов итераторов.

Таблица 7. Характеристики итераторов

Тип итератора	Запись/Чтение	Хранение значения	Направление	Доступ
Со случайным доступом	Запись и чтение	Возможно	Оба направления	Случайный
Двунаправленный	Запись и чтение	Возможно	Оба направления	Линейный
Прямой	Запись и чтение	Возможно	Только прямое	Линейный
Выходной	Только запись	Невозможно	Только прямое	Линейный
Входной	Только чтение	Невозможно	Только прямое	Линейный

Возможные проблемы с STL

Шаблоны классов STL достаточно сложны, это предъявляет высокие требования к компилятору. Не все компиляторы выдерживают. Рассмотрим некоторые возможные проблемы.

Во-первых, иногда бывает сложно отыскать ошибку, поскольку компилятор сообщает, что она произошла где-то в глубинах заголовочного файла, в то время как на самом деле она произошла на самом видном месте в исходном файле. Вам придется перелопатить уйму кода, комментируя одну строчку за другой, а диагностировать ошибку, возможно, так и не удастся.

Прекомпиляция заголовочных файлов, которая невероятно ускоряет процесс компиляции программы, может вызвать определенные проблемы при использовании STL. Если вам кажется, что что-то не так, попробуйте отключить прекомпиляцию.

STL может генерировать разные забавные сообщения. Например, «При преобразовании могут потеряться значащие разряды!». В принципе, эти ошибки довольно безобидны, не стоит обращать на них внимания. Если они вас сильно раздражают, можно их отключить вообще.

Несмотря на мелкие претензии, которые можно предъявить к STL, это удивительно мощный и гибкий инструмент. Все ошибки можно исключить на этапе компиляции, а не во время работы программы. Алгоритмы и контейнеры имеют очень содержательный и устойчивый интерфейс. То, что работает в применении к одному контейнеру или алгоритму, будет, скорее всего, работать и в применении к другим (конечно, при условии их правильного использования).