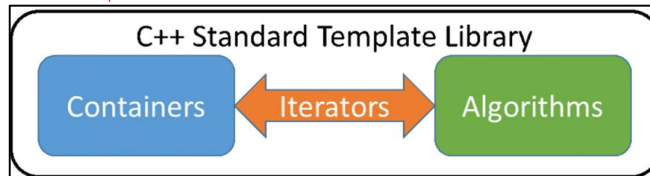


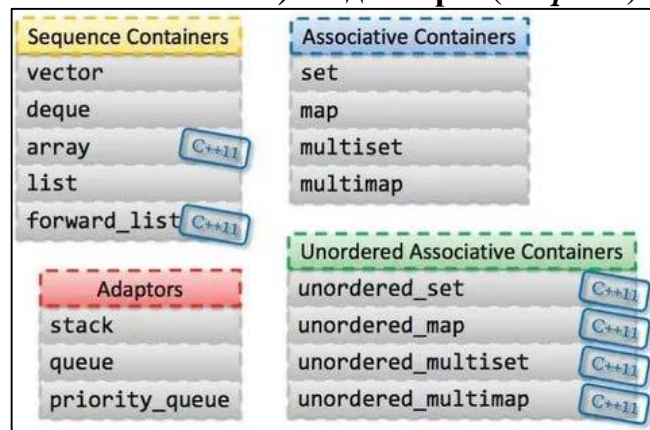
# ЛАБОРАТОРНАЯ РАБОТА №5 ПО ПРЕДМЕТУ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ». ЧАСТЬ 2. «ВВЕДЕНИЕ В СТАНДАРТНУЮ БИБЛИОТЕКУ ШАБЛОНОВ. ПОСЛЕДОВАТЕЛЬНЫЕ И АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ»



Библиотека *STL* (*Standard Template Library*) вошла в стандарты C++ 1998 и 2003 годов (*ISO/IEC 14882:1998* и *ISO/IEC 14882:2003*) и с тех пор считается одной из составных частей стандартной библиотек языка C++. Библиотека *STL* предоставляет шаблонные классы и функции общего назначения, которые реализуют многие популярные и часто используемые алгоритмы и структуры данных. Например, она включает поддержку векторов, списков, очередей, стеков, определяет различные функции, обеспечивающие к ним доступ. Синтаксис библиотеки *STL* основан на использовании таких синтаксических конструкций языка C++ как шаблоны (*templates*) классов и шаблоны функций.

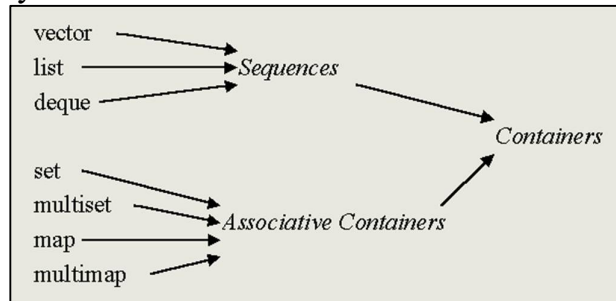
**Библиотека *STL*** представляет собой набор шаблонных классов и функций общего назначения. Ядро стандартной библиотеки шаблонов включает три основных элемента: контейнеры, алгоритмы, итераторы. Они работают совместно друг с другом, предоставляя тем самым готовые решения для различных задач.

**Контейнеры** — это объекты, которые содержат в себе совокупность других объектов. Контейнеры можно разделить на три категории: **последовательные контейнеры** (*sequence containers*), **ассоциативные контейнеры** (*associative containers*) и **адаптеры** (*adaptors*) контейнеров.



Например, класс *vector* определяет динамический массив, класс *queue* создает двустороннюю очередь, а класс *list* обеспечивает работу с линейным списком. Эти контейнеры называются **последовательными контейнерами** и являются базовыми в *STL*.

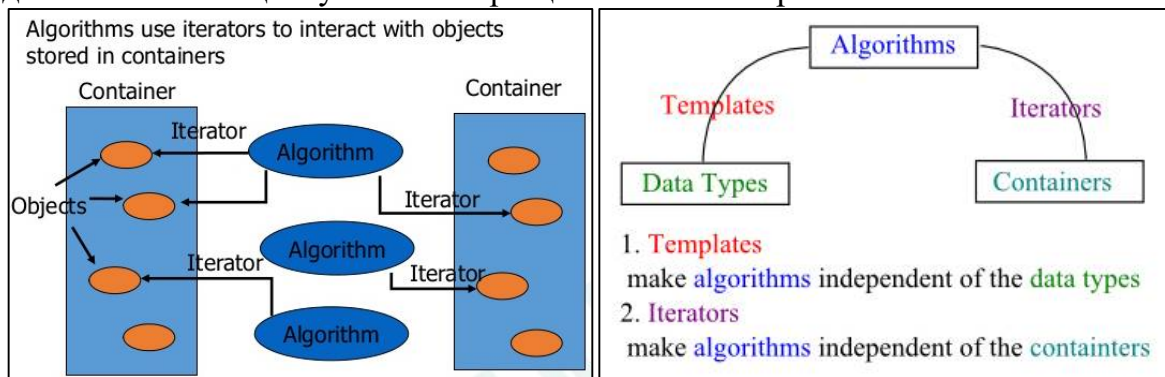
Помимо базовых, библиотека *STL* определяет **ассоциативные контейнеры**, которые позволяют эффективно находить нужные значения на основе ключей. Например, класс *map* обеспечивает хранение пар "ключ-значение" и предоставляет возможность находить значение по заданному уникальному ключу.



Каждый контейнерный класс определяет набор методов, которые можно применять к данному контейнеру и его элементам. Например, список (*list*) включает методы, предназначенные для выполнения вставки, удаления, объединения элементов.

**Алгоритмы** представляют собой функции, которые обрабатывают содержимое контейнеров. Их возможности включают средства инициализации, сортировки, поиска, преобразования содержимого контейнеров. Многие алгоритмы работают с заданным диапазоном элементов контейнера.

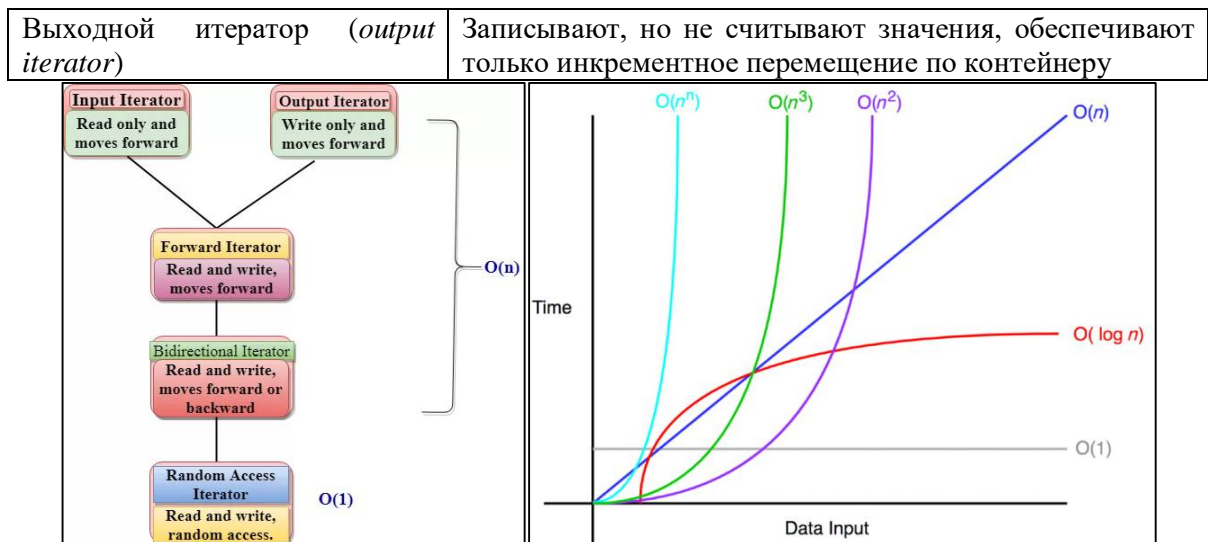
**Итераторы** действуют подобно указателям. Они позволяют циклически опрашивать содержимое контейнера практически так же, как это делается с помощью указателя при циклическом опросе элементов массива.



Существует пять типов итераторов (таблица №1).

Таблица №1 – Итераторы библиотеки *STL*

Наименование итератора	Описание итератора
Итератор произвольного доступа ( <i>random access iterator</i> )	Сохраняют и считывают значения, позволяют организовать произвольный доступ к элементам контейнера
Двунаправленный итератор ( <i>bidirectional iterator</i> )	Сохраняют и считывают значения, обеспечивают инкрементно-декрементное перемещение по контейнеру
Однонаправленный итератор ( <i>forward iterator</i> )	Сохраняют и считывают значения, обеспечивают только инкрементное перемещение по контейнеру
Входной итератор ( <i>input iterator</i> )	Считывают, но не записывают значения, обеспечивают только инкрементное перемещение по контейнеру



Итераторы обрабатываются аналогично указателям. Их можно инкрементировать и декрементировать. К ним можно применять оператор разыменования адреса \*. Итераторы объявляются с помощью типа *iterator*, определяемого различными контейнерами.

Библиотека *STL* поддерживает **реверсивные итераторы**, которые являются либо двунаправленными, либо итераторами произвольного доступа, позволяя перемещаться по последовательности в обратном направлении. Следовательно, если реверсивный итератор указывает на конец последовательности, то после инкрементирования он будет указывать на элемент, расположенный перед концом последовательности.

При ссылке на различные типы итераторов в описаниях шаблонов в библиотеке *STL* используются термины, описанные в таблице №2.

Таблица №2 – Термины, используемые для ссылок на различные типы итераторов

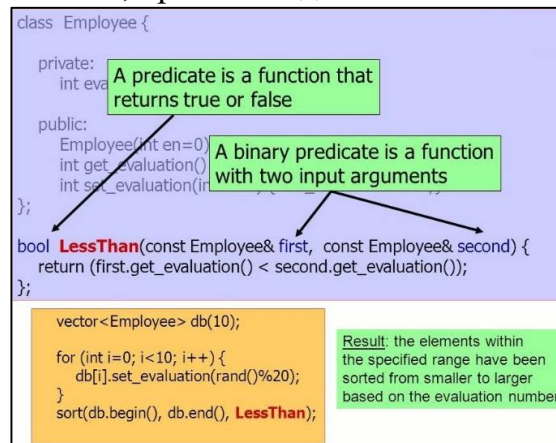
Термин	Представляемый итератор
<i>BiIter</i>	Двунаправленный итератор
<i>ForBiIter</i>	Однонаправленный итератор
<i>InIter</i>	Входной итератор
<i>OutIter</i>	Выходной итератор
<i>RandIter</i>	Итератор произвольного доступа

*STL* опирается не только на контейнеры, алгоритмы и итераторы, но и на другие стандартные компоненты. Основными из них являются распределители памяти, предикаты, функции сравнения.

**Распределитель памяти (*allocator*)** управляет выделением памяти для контейнера. Каждый контейнер имеет свой распределитель памяти. Стандартный распределитель — объект класса *allocator*, но при необходимости можно определять собственные распределители. В большинстве случаев стандартного распределителя достаточно.

**Предикат (*predicate*)** – функция, возвращающая в качестве результата значение ИСТИНА/ЛОЖЬ. Некоторые алгоритмы и контейнеры используют

специальный тип функции, называемый предикатом. Существует два варианта предикатов: унарный и бинарный. Унарный предикат принимает один аргумент, а бинарный — два. Эти функции возвращают значения ИСТИНА/ЛОЖЬ, но точные условия, которые заставят их вернуть истинное или ложное значение, определяются программистом. В бинарном предикате аргументы всегда расположены в порядке первый, второй относительно функции, которая вызывает этот предикат. Как для унарного, так и для бинарного предикатов аргументы должны содержать значения, тип которых совпадает с типом объектов, хранимых данным контейнером.



**Функции сравнения** сравнивают два элемента последовательности. Некоторые алгоритмы и классы используют специальный тип бинарного предиката, который сравнивает два элемента. Функции сравнения возвращают значение *true*, если их первый аргумент меньше второго. Функции сравнения идентифицируются с помощью типа *Comp*.

Помимо заголовков, требуемых различными классами *STL*, стандартная библиотека C++ включает заголовки `<utility>` и `<functional>`, которые обеспечивают поддержку *STL*. Например, в заголовке `<utility>` определяется шаблонный класс *pair*, который может хранить пару ключ-значение. Шаблоны в заголовке `<functional>` позволяют создавать объекты, которые определяют функцию *operator()*. Эти объекты называются объектами-функциями, и их во многих случаях можно использовать вместо указателей на функции. Существует несколько встроенных объектов-функций, объявленных в заголовке `<functional>`: *plus()*, *minus()*, *multiplies()*, *divides()*, *modulus()*, *negate()*, *equal\_to()*, *not\_equal\_to()*, *greater()*, *greater\_equal()*, *less()*, *less\_equal()*, *logical\_and()*, *logical\_or()*, *logical\_not()*.

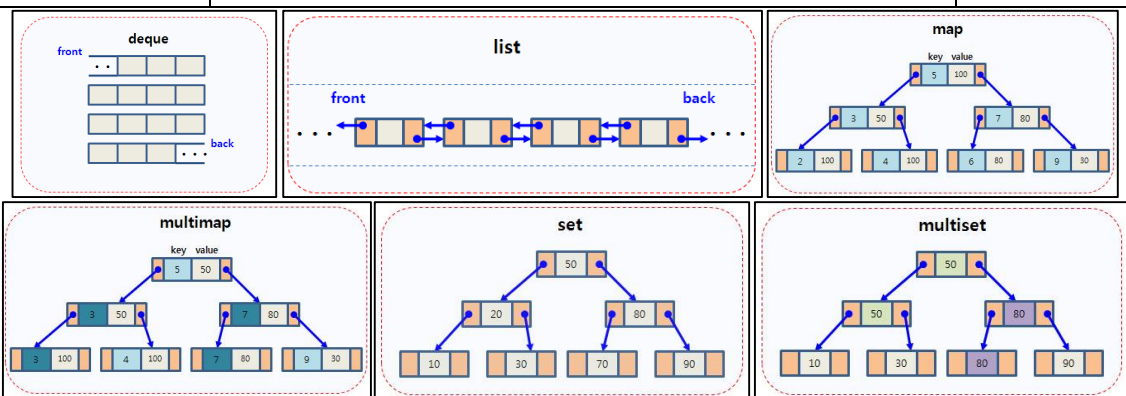
Наиболее широко используется объект-функция *less*, которая определяет, при каких условиях один объект меньше другого. Объекты-функции можно использовать вместо реальных указателей на функции в алгоритмах *STL*. Используя объекты-функции вместо указателей на функции, библиотека *STL* в некоторых случаях генерирует более эффективный программный код.

## КОНТЕЙНЕРНЫЕ КЛАССЫ

Контейнеры представляют собой объекты *STL*, которые предназначены для хранения данных. Класс *string* так же является контейнером. Контейнеры, определяемые в библиотеке *STL*, представлены в таблице №3.

Таблица №3 – Основные контейнеры библиотеки *STL*

Наименование контейнера	Описание контейнера	Требуемый заголовок
<i>vector</i>	Динамический массив	<code>&lt;vector&gt;</code>
<i>stack</i>	Стек	<code>&lt;stack&gt;</code>
<i>list</i>	Линейный список	<code>&lt;list&gt;</code>
<i>queue</i>	Очередь	<code>&lt;queue&gt;</code>
<i>priority_queue</i>	Приоритетная очередь	<code>&lt;queue&gt;</code>
<i>deque</i>	Дек (двусторонняя очередь)	<code>&lt;deque&gt;</code>
<i>set</i>	Множество, в котором каждый элемент уникален	<code>&lt;set&gt;</code>
<i>bitset</i>	Битовое множество, в котором каждый элемент уникален	<code>&lt;bitset&gt;</code>
<i>multiset</i>	Мультимножество. Множество, в котором каждый элемент необязательно уникален	<code>&lt;set&gt;</code>
<i>map</i>	Отображение. Хранит пары «ключ-значение», в которых каждый ключ связан только с одним значением	<code>&lt;map&gt;</code>
<i>multimap</i>	Мультиотображение. Хранит пары «ключ-значение», в которых каждый ключ может быть связан с двумя или более значениями	<code>&lt;map&gt;</code>



Поскольку имена типов в объявлениях шаблонных классов произвольны, контейнерные классы объявляют *typedef*-версии этих типов, что конкретизирует имена типов. Некоторые из наиболее популярных *typedef*-имен приведены в таблице №4.

Таблица №4 – *Typedef*-версии имен типов шаблонных классов

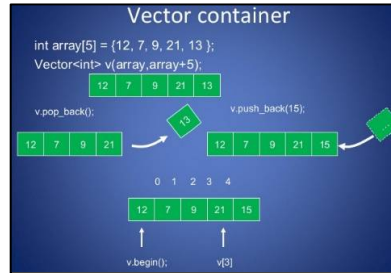
Наименование типа	Характеристика имени
<i>size_type</i>	Целочисленный тип. Аналогичен типу <i>size_t</i>
<i>reference</i>	Ссылка на элемент
<i>const_reference</i>	Константная ( <i>const</i> ) ссылка на элемент
<i>iterator</i>	Итератор
<i>const_iterator</i>	Константный ( <i>const</i> ) итератор
<i>reverse_iterator</i>	Риверсивный итератор
<i>const_reverse_iterator</i>	Константный риверсивный итератор
<i>value_type</i>	Тип значения, хранимого в контейнере (то же самое, что и



	обобщенный тип $T$ )
<i>allocator_type</i>	Тип распределения памяти
<i>key_type</i>	Тип ключа
<i>key_compare</i>	Тип функции, которая сравнивает два ключа
<i>mapped_type</i>	Тип значения, сохраняемого в отображении (то же самое, что и обобщенный тип $T$ )
<i>value_compare</i>	Тип функции, которая сравнивает два значения

## КОНТЕЙНЕР VECTOR

Векторы представляют собой динамические массивы. Класс *vector* поддерживает динамический массив, который при необходимости может увеличивать свой размер.



Размер статического массива **фиксируется** во время компиляции. И хотя это самый эффективный способ реализации массивов, он в то же время является и самым ограничивающим, поскольку размер массива нельзя изменять во время выполнения программы. Эта проблема решается с помощью вектора, который по мере необходимости обеспечивает выделение дополнительного объема памяти. Несмотря на то, что вектор — это динамический массив, тем не менее, для доступа к его элементам можно использовать стандартное обозначение индексации массивов.

Шаблонная спецификация для класса *vector*:

```
template <class Type, class Allocator = allocator<Type>>
class vector
```

Здесь параметр *Type* определяет тип сохраняемых данных, *Allocator* означает распределитель памяти, который по умолчанию использует стандартный распределитель. Класс *vector* имеет следующие конструкторы:

```
vector();
explicit vector(const Allocator& allocator);
explicit vector(size_type count);
vector(size_type count, const Type& value);
vector(size_type count, const Type& value, const Allocator&
allocator);
vector(const vector& source);
vector(vector&& source);
vector(initializer_list<Type> init_list, const Allocator&
allocator);
template <class InputIterator>
vector(InputIterator first, InputIterator last);
template <class InputIterator>
vector(InputIterator first, InputIterator last, const Allocator&
allocator);
```

Первая форма конструктора предназначена для создания пустого вектора. Параметр *allocator* в конструкторах определяет класс распределителя для использования с объектом создаваемого вектора. Параметр *count* определяет количество элементов в создаваемом векторе. Параметр *value* определяет значение элементов в создаваемом векторе. Параметр *source* определяет вектор, копией которого должен быть создаваемый вектор. Параметр *first* определяет положение первого элемента в диапазоне копируемых элементов. Параметр *last* определяет положение первого элемента за пределами диапазона копируемых элементов. Параметр *init\_list* определяет список инициализации, содержащий элементы для копирования.

Ключевое слово *explicit* запрещает автоматическое создание конвертирующего конструктора, т.е. используется для создания «неконвертирующих конструкторов» (*non converting constructors*).

```
Example e = Example(0, 50); // Explicit call

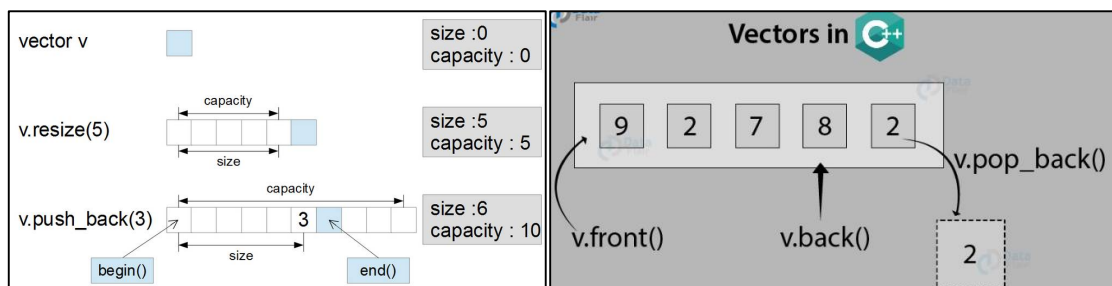
Example e(0, 50);           // Implicit call
```

Для достижения максимальной гибкости и переносимости любой объект, который предназначен для хранения в векторе, должен иметь конструктор по умолчанию. Кроме того, он должен определять перегрузку операторов "<" и "==" Некоторые компиляторы могут потребовать определения и других операторов сравнения. Все встроенные типы данных автоматически удовлетворяют этим требованиям.

Рассмотрим несколько примеров объявлений вектора.

```
vector<int> iv1; //Создание вектора нулевой длины для хранения
int-значений
vector<char> cv1(5); //Создание 5-элементного вектора для
хранения char-значений
vector<char> cv2(5, 'x'); //Инициализация 5-элементного char
вектора со значениями 'x'
vector<int> iv2(iv1); //Создание int-вектора на основе int
вектора iv1
```

Для класса *vector* определены следующие операторы сравнения: ==, <, <=, !=, > и >=. Для вектора также определен оператор индексации "[]", который позволяет получить доступ к элементам вектора с помощью стандартной записи с использованием индексов.



Векторы при необходимости увеличивают свой размер, поэтому можно определять его величину во время работы программы, а не только во время компиляции. Методы-элементы контейнера *vector* приведены в таблице №5.

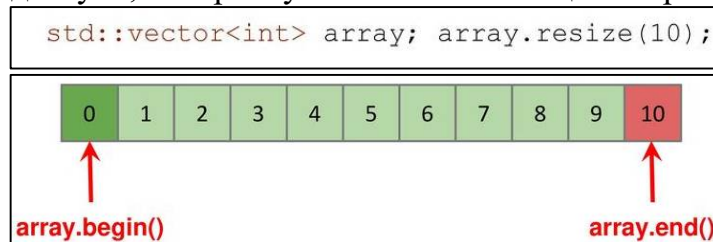
Таблица №5 – Методы контейнера *vector*

Сигнаutra метода	Назначение метода
<code>void <b>assign</b>(size_type count, const Type&amp; value);</code> <code>void <b>assign</b>(initializer_list&lt;Type&gt; init_list);</code> <code>template &lt;class InputIterator&gt;</code> <code>void <b>assign</b>(InputIterator first, InputIterator last);</code>	Стирает вектор и копирует указанные элементы в пустой вектор
<code>reference <b>at</b>(size_type position);</code> <code>const_reference <b>at</b>(size_type position) const;</code>	Возвращает ссылку на элемент в указанном месте вектора (параметр <i>position</i> )
<code>reference <b>back</b>();</code> <code>const_reference <b>back</b>() const;</code>	Возвращает ссылку на последний элемент в векторе
<code>const_iterator <b>begin</b>() const;</code> <code>iterator <b>begin</b>();</code>	Возвращает итератор произвольного доступа, указывающий на первый элемент вектора
<code>size_type <b>capacity</b>() const;</code>	Возвращает количество элементов, которые вектор может содержать, не выделяя больше места для хранения
<code>void <b>clear</b>();</code>	Удаляет все элементы вектора
<code>bool <b>empty</b>() const;</code>	Проверяет, является ли вектор пустым
<code>iterator <b>end</b>();</code> <code>const_iterator <b>end</b>() const;</code>	Возвращает итератор на конец вектора
<code>iterator <b>erase</b>(const_iterator position);</code> <code>iterator <b>erase</b>(const_iterator first, const_iterator last);</code>	Удаляет элемент или диапазон элементов вектора из заданных позиций. Возвращает итератор для элемента, расположенного после удаленного
<code>reference <b>front</b>();</code> <code>const_reference <b>front</b>() const;</code>	Возвращает ссылку на первый элемент вектора
<code>allocator_type <b>get_allocator</b>() const;</code>	Возвращает копию объекта распределителя, используемого для построения вектора
<code>iterator <b>insert</b>(const_iterator position, const Type&amp; value);</code> <code>iterator <b>insert</b>(const_iterator position, Type&amp;&amp; value);</code> <code>void <b>insert</b>(const_iterator position, size_type count, const Type&amp; value);</code> <code>template &lt;class InputIterator&gt;</code> <code>void <b>insert</b>(const_iterator position, InputIterator first, InputIterator last);</code>	Вставляет непосредственно перед элементом элемент, ряд элементов или диапазон элементов в вектор в заданном положении
<code>size_type <b>max_size</b>() const;</code>	Возвращает максимальную длину вектора.

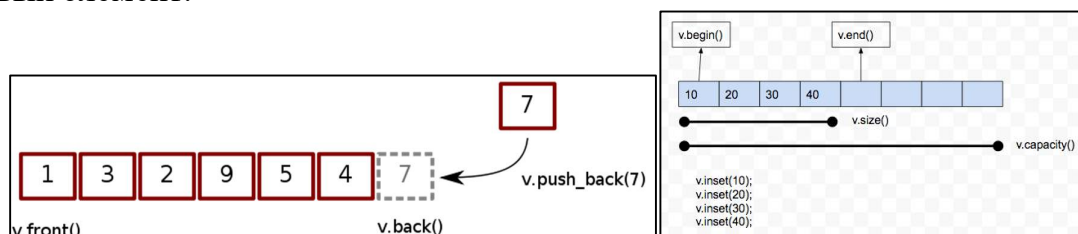


	Это максимальный потенциальный размер, которого может достичь контейнер из-за известных ограничений реализации системы или библиотеки.
<i>reference operator[]</i> (size_type position); <i>const_reference operator[]</i> (size_type position) const;	Возвращает ссылку на элемент вектора в заданной позиции
<i>void pop_back</i> ();	Удаляет элемент в конце вектора
<i>void push_back</i> (const T& value); <i>void push_back</i> (T&& value);	Добавляет элемент в конец вектора
<i>reverse_iterator rbegin</i> (); <i>const_reverse_iterator rbegin</i> () const;	Возвращает итератор на первый элемент в обратном векторе
<i>reverse_iterator rend</i> (); <i>const_reverse_iterator rend</i> () const;	Возвращает итератор, который указывает на следующий за последним элементом в обратном векторе. Т.е. возвращает реверсивный итератор для начала вектора
<i>void reserve</i> (size_type num);	Резервирует минимальную длину хранилища (указанную в качестве параметра <i>num</i> ) для векторного объекта, выделяя при необходимости место
<i>void resize</i> (size_type new_size); <i>void resize</i> (size_type new_size, Type value);	Задаст новый размер для вектора. Список инициализации ( <i>value</i> ) добавляется к вектору, если новый размер больше исходного размера. Если значение <i>value</i> не указано, то новые объекты используют свой конструктор по умолчанию.
<i>size_type size</i> () const;	Возвращает количество элементов в векторе
<i>void swap</i> (vector<Type, Allocator>& right); <i>friend void swap</i> (vector<Type, Allocator>& left, vector<Type, Allocator>& right);	Меняет местами элементы двух векторов

Функция *begin()* возвращает итератор произвольного доступа, который указывает на начало вектора. Функция *end()* возвращает итератор произвольного доступа, который указывает на конец вектора.



Функция *push\_back()* помещает заданное значение в конец вектора. При необходимости длина вектора увеличивается так, чтобы он мог принять новый элемент.



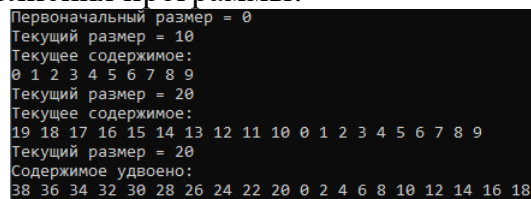
С помощью метода *insert()* можно добавлять элементы в середину вектора. Для доступа к элементам вектора и их модификации можно использовать средство индексации массивов. А с помощью метода *erase()* можно удалять элементы из вектора.

Рассмотрим пример, который иллюстрирует базовое поведение вектора.

*//Пример №1. Демонстрация базового поведения контейнера «вектор»*

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    vector<int> v; //создание вектора нулевой длины
    unsigned int i;
    cout << "Первоначальный размер = " << v.size() << endl;
    //Помещаем значения в конец вектора
    for (i = 0; i < 10; i++) v.push_back(i);
    cout << "Текущий размер = " << v.size() << endl;
    cout << "Текущее содержимое:\n";
    for (i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    //Помещаем в начало вектора новые значения
    for (i = 0; i < 10; i++) v.insert(v.begin(), i+10);
    cout << "Текущий размер = " << v.size() << endl;
    cout << "Текущее содержимое:\n";
    for (i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    //Изменяем содержимое вектора
    for (i = 0; i < v.size(); i++) v[i] = v[i] + v[i];
    cout << "Текущий размер = " << v.size() << endl;
    cout << "Содержимое удвоено:\n";
    for (i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

Результат выполнения программы:



```
Первоначальный размер = 0
Текущий размер = 10
Текущее содержимое:
0 1 2 3 4 5 6 7 8 9
Текущий размер = 20
Текущее содержимое:
10 11 12 13 14 15 16 17 18 19 0 1 2 3 4 5 6 7 8 9
Текущий размер = 20
Содержимое удвоено:
38 36 34 32 30 28 26 24 22 20 0 2 4 6 8 10 12 14 16 18
```

В функции *main()* создается вектор *v* для хранения *int*-элементов. Поскольку при его создании не было предусмотрено никакой инициализации, вектор *v* получился пустым, а его емкость равна нулю. Т.е. создан вектор нулевой длины. Это подтверждается вызовом функции-элемента *size()*. Затем, используя функцию-элемент *push\_back()*, в конец

этого вектора помещаются 10 элементов, что заставляет вектор увеличиться в размере, чтобы разместить новые элементы. Теперь размер вектора стал равным 10. Для отображения содержимого вектора *v* используется стандартная запись индексации массивов. После этого в начало вектора добавляются еще 10 элементов, и вектор *v* автоматически увеличивается в размере, чтобы и их принять на хранение. Далее используя стандартную запись индексации массивов, изменяются значения элементов вектора *v*.

Для управления циклами, используемыми для отображения содержимого вектора *v* и его модификации, в качестве признака их завершения применяется значение размера вектора, получаемое с помощью функции *v.size()*. Одно из преимуществ векторов перед массивами состоит в том, что есть возможность узнать текущий размер вектора.

## ДОСТУП К СОДЕРЖИМОМУ ВЕКТОРА С ПОМОЩЬЮ ИТЕРАТОРА

Массивы и указатели в C++ тесно связаны между собой. К элементам массива можно получить доступ как с помощью индекса, так и с помощью указателя. В библиотеке *STL* аналогичная связь существует между векторами и итераторами. Это значит, что к элементам вектора можно обращаться как с помощью индекса, так и с помощью итератора. Эта возможность демонстрируется в следующей программе.

```
//Пример №2. Доступ к вектору с помощью итератора
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<char> v; //создание вектора нулевой длины
    int i;
    for (i = 0; i < 10; i++) v.push_back('A' + i);
    //доступ к содержимому вектора с помощью индекса
    for (i = 0; i < 10; i++) cout << v[i] << " "; cout << endl;
    //доступ к содержимому вектора с помощью итератора
    vector<char>::iterator p = v.begin();
    while (p != v.end()) {
        cout << *p << " ";
        p++;
    }
    return 0;
}
```

Результат выполнения этой программы.

A	B	C	D	E	F	G	H	I	J
A	B	C	D	E	F	G	H	I	J

В этой программе сначала создается вектор нулевой длины. Затем с помощью функции *push\_back()* в конец вектора помещаются символы, в результате чего размер вектора увеличивается. **Тип итератора определяется контейнерными классами.** В программе итератор *p* инициализируется таким образом, чтобы он указывал на начало вектора (с помощью метода

*begin()*). Итератор, который возвращает эта функция, можно затем использовать для поэлементного доступа к элементам вектора, инкрементируя его. Этот процесс аналогичен тому, как можно использовать указатель для доступа к элементам массива. Чтобы определить, когда будет достигнут конец вектора, используется метод-элемент *end()*. Этот метод возвращает итератор, установленный за последним элементом вектора. Поэтому, если значение *p* равно *v.end()*, значит, конец вектора достигнут.

### ВСТАВКА И УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ВЕКТОРА

Помимо занесения новых элементов в конец вектора, есть возможность вставлять элементы в середину вектора, используя функцию *insert()*. Удалять элементы можно с помощью функции *erase()*. Использование функций *insert()* и *erase()* демонстрируется в следующей программе.

```
//Пример №3. Вставка и удаление элементов вектора
#include <iostream>
#include <vector>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    vector<char> v;
    unsigned int i;
    for (i = 0; i < 10; i++) v.push_back('A' + i);
    //Отображаем исходное содержимое вектора.
    cout << "Размер = " << v.size() << endl;
    cout << "Исходное содержимое вектора:\n";
    for (i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl << endl;
    vector<char>::iterator p = v.begin();
    p += 2; //перемещаем указатель на 3-й элемент вектора
    //Вставляем 10 символов 'X' в вектор v.
    v.insert(p, 10, 'X');
    //Отображаем содержимое вектора после вставки символов
    cout << "Размер вектора после вставки = " << v.size() << endl;
    cout << "Содержимое вектора после вставки:\n";
    for (i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl << endl;
    //Удаление вставленных элементов.
    p = v.begin();
    p += 2; //перемещаем указатель на 3-й элемент вектора
    v.erase(p, p + 10); //Удаляем 10 элементов подряд
    //Отображаем содержимое вектора после удаления символов
    cout << "Размер вектора после удаления = " << v.size() << endl;
    cout << "Содержимое вектора после удаления символов:\n";
    for (i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

Результаты работы программы.

```
Размер = 10
Исходное содержимое вектора:
A B C D E F G H I J

Размер вектора после вставки = 20
Содержимое вектора после вставки:
A B ! ! ! ! ! ! ! ! C D E F G H I J

Размер вектора после удаления = 10
Содержимое вектора после удаления символов:
A B C D E F G H I J
```

## ХРАНЕНИЕ В ВЕКТОРЕ ОБЪЕКТОВ КЛАССА

Векторы могут служить не только для хранения значений встроенных типов, но и для хранения объектов любого типа, включая объекты классов. Рассмотрим пример, в котором вектор используется для хранения объектов класса *three\_d*. В этом классе определяются конструктор по умолчанию и перегруженные версии операторов "<" и "==". Возможно придется определить и другие операторы сравнения. Это зависит от того, как используемый компилятор реализует библиотеку *STL*.

//Пример №4. Хранение в векторе объектов класса

```
#include <iostream>
#include <vector>
using namespace std;
class Point3D {
    int x, y, z;
public:
    Point3D() { x = y = z = 0; }
    Point3D(int a, int b, int c) { x = a; y = b; z = c; }
    Point3D& operator+(int a) {
        x += a;
        y += a;
        z += a;
        return *this;
    }
    friend ostream& operator<<(ostream& stream, Point3D obj);
    friend bool operator<(Point3D a, Point3D b);
    friend bool operator==(Point3D a, Point3D b);
};
//Отображаем координаты X, Y, Z с помощью оператора вывода
ostream& operator<<(ostream& stream, Point3D obj) {
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream;
}
bool operator<(Point3D a, Point3D b) {
    return (a.x + a.y + a.z) < (b.x + b.y + b.z);
}
bool operator==(Point3D a, Point3D b) {
    return (a.x + a.y + a.z) == (b.x + b.y + b.z);
}
int main() {
    vector<Point3D> v;
```



```

unsigned int i;
//Добавляем в вектор объекты.
for (i = 0; i < 10; i++)
    v.push_back(Point3D(i, i + 2, i - 3));
//Отображаем содержимое вектора.
for (i = 0; i < v.size(); i++)
    cout << v[i];
cout << endl;
//Модифицируем объекты в векторе.
for (i = 0; i < v.size(); i++) v[i] = v[i] + 10;
//Отображаем содержимое модифицированного вектора.
for (i = 0; i < v.size(); i++) cout << v[i];
return 0;
}

```

Результат работы программы:

```

0, 2, -3
1, 3, -2
2, 4, -1
3, 5, 0
4, 6, 1
5, 7, 2
6, 8, 3
7, 9, 4
8, 10, 5
9, 11, 6

10, 12, 7
11, 13, 8
12, 14, 9
13, 15, 10
14, 16, 11
15, 17, 12
16, 18, 13
17, 19, 14
18, 20, 15
19, 21, 16

```

Векторы обеспечивают безопасность хранения элементов, предоставляя гибкость их обработки, но уступают массивам в эффективности использования. Поэтому для большинства задач программирования чаще отдается предпочтение обычным массивам. Однако возможны ситуации, когда уникальные особенности векторов оказываются важнее затрат системных ресурсов, связанных с их использованием.

Многие функции библиотеки *STL* используют итераторы. Этот факт позволяет выполнять операции с двумя контейнерами одновременно.

Рассмотрим, например, такой формат векторного метода *insert()*.

```

template <class InputIterator>
void insert(const_iterator position, InputIterator first,
InputIterator last);

```

Этот метод вставляет исходную последовательность, определенную итераторами *first* и *last*, в принимаемую последовательность, начиная с позиции *position*. При этом нет никаких требований, чтобы итератор *position* относился к тому же вектору, с которым связаны итераторы *first* и *last*. Таким образом, используя эту версию метода *insert()*, можно один вектор вставить в другой.

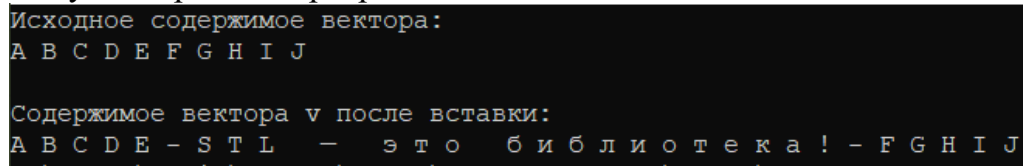
//Пример №5. Использование вставки содержимого одного вектора в другой

```

#include <iostream>
#include <vector>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    vector<char> v1, v2;
    unsigned int i;
    for (i = 0; i < 10; i++) v1.push_back('A' + i);
    //Отображаем исходное содержимое вектора.
    cout << "Исходное содержимое вектора:\n";
    for (i = 0; i < v1.size(); i++) cout << v1[i] << " ";
    cout << endl << endl;
    //Инициализируем второй вектор.
    char str[] = "-STL – это библиотека!-";
    for (i = 0; str[i]; i++) v2.push_back(str[i]);
    //итераторы для середины вектора v, начала и конца вектора v2
    vector<char>::iterator p = v1.begin() + 5;
    vector<char>::iterator p2start = v2.begin();
    vector<char>::iterator p2end = v2.end();
    //Вставляем вектор v2 в вектор v.
    v1.insert(p, p2start, p2end);
    //Отображаем результат вставки
    cout << "Содержимое вектора v после вставки:\n";
    for (i = 0; i < v1.size(); i++) cout << v1[i] << " ";
    return 0;
}

```

Результат работы программы:



```

Исходное содержимое вектора:
A B C D E F G H I J

Содержимое вектора v после вставки:
A B C D E - S T L - э т о б и б л и о т е к а ! - F G H I J

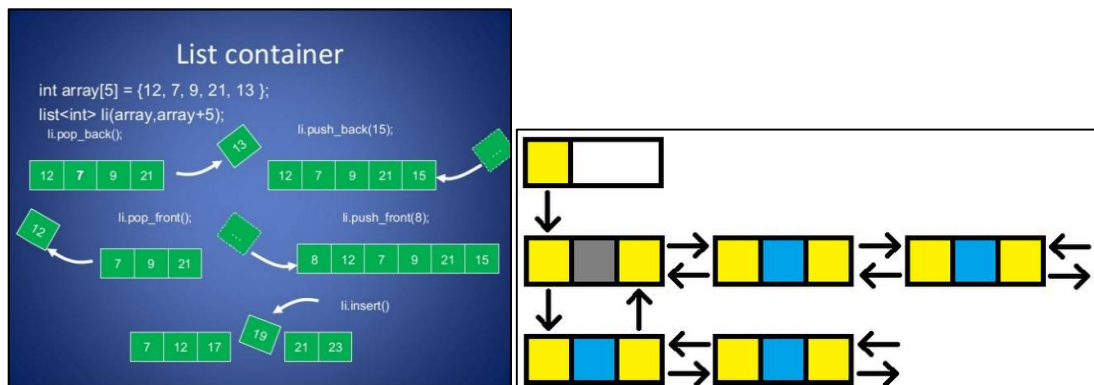
```

Содержимое вектора v2 вставлено в середину вектора v1.

Итераторы являются связующими средствами, которые делают библиотеку *STL* единым целым. Они позволяют работать с двумя (и больше) объектами *STL* одновременно.

## КОНТЕЙНЕР LIST

**Список** является контейнером с двунаправленным последовательным доступом к элементам. Класс *list* поддерживает функционирование двунаправленного линейного списка. В отличие от вектора, в котором реализована поддержка произвольного доступа, **список позволяет получать к своим элементам только последовательно**.



**Двунаправленность** списка означает, что доступ к его элементам возможен в двух направлениях: от начала к концу и от конца к началу. Шаблонная спецификация класса *list* выглядит следующим образом.

```
template <class Type, class Allocator= allocator<Type>>
class list
```

Здесь *Type* — тип данных, сохраняемых в списке, а элемент *allocator* означает распределитель памяти, который по умолчанию использует стандартный распределитель. В классе *list* определены следующие конструкторы.

```
list();
explicit list(const Allocator& Al);
explicit list(size_type Count);
list(size_type Count, const Type& Val);
list(size_type Count, const Type& Val, const Allocator& Al);
list(const list& Right);
list(list&& Right);
list(initializer_list<Type> IList, const Allocator& Al);
template <class InputIterator>
list(InputIterator First, InputIterator Last);
template <class InputIterator>
list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Где параметр *Al* определяет класс распределителя для использования с этим объектом, параметр *Count* определяет количество элементов в создаваемом списке, параметр *Val* определяет значение элементов в списке, параметр *Right* определяет список, из которого создается список, параметр *First* определяет положение первого элемента в диапазоне копируемых элементов, параметр *Last* определяет положение первого элемента за пределами диапазона копируемых элементов, параметр *IList* определяет список инициализации, содержащий элементы для копирования.

Методы-элементы, определенные в классе *list*, перечислены в таблице №6. В конец списка, как и в конец вектора, элементы можно помещать с помощью метода *push\_back()*, но с помощью метода *push\_front()* можно помещать элементы и в начало списка. Элемент можно также

вставить и в середину списка, для этого используется функция *insert()*. Один список можно поместить в другой, используя функцию *splice()*. С помощью метода *merge()* два списка можно объединить и упорядочить результат.

Таблица №6 – Методы класса *list*

Сигнаutra метода	Назначение метода
<code>void <b>assign</b>(size_type Count, const Type&amp; Val);</code> <code>void <b>assign</b> initializer_list&lt;Type&gt; IList);</code> <code>template &lt;class InputIterator&gt;</code> <code>void <b>assign</b>(InputIterator First, InputIterator Last);</code>	Стирает элементы из списка и копирует новый набор элементов в целевой список.
<code>reference <b>back</b>();</code> <code>const_reference <b>back</b>() const;</code>	Возвращает ссылку на последний элемент списка
<code>const_iterator <b>begin</b>() const;</code> <code>iterator <b>begin</b>();</code>	Возвращает итератор, адресуящий первый элемент в списке
<code>void <b>clear</b>();</code>	Удаляет все элементы списка
<code>bool <b>empty</b>() const;</code>	Проверяет, пуст ли список
<code>const_iterator <b>end</b>() const;</code> <code>iterator <b>end</b>();</code>	Возвращает итератор, который обращается к местоположению, следующему за последним элементом в списке
<code>iterator <b>erase</b>(iterator Where);</code> <code>iterator <b>erase</b>(iterator first, iterator last);</code>	Удаляет элемент или диапазон элементов в списке из заданных позиций
<code>reference <b>front</b>();</code> <code>const_reference <b>front</b>() const;</code>	Возвращает ссылку на первый элемент списка
<code>Allocator <b>get_allocator</b>() const;</code>	Возвращает копию объекта распределителя, используемого для построения списка
<code>iterator <b>insert</b>(iterator Where, const Type&amp; Val);</code> <code>iterator <b>insert</b>(iterator Where, Type&amp;&amp; Val);</code> <code>void <b>insert</b>(iterator Where, size_type Count, const Type&amp; Val);</code> <code>iterator <b>insert</b>(iterator Where, initializer_list&lt;Type&gt; IList);</code> <code>template &lt;class InputIterator&gt;</code> <code>void <b>insert</b>(iterator Where, InputIterator First, InputIterator Last);</code>	Вставляет элемент или ряд элементов или диапазон элементов в список в заданной позиции
<code>size_type <b>max_size</b>() const;</code>	Возвращает максимальную длину списка
<code>void <b>pop_back</b>();</code>	Удаляет элемент в конце списка
<code>void <b>pop_front</b>();</code>	Удаляет элемент в начале списка
<code>void <b>push_back</b>(void push_back(Type&amp;&amp; val);</code>	Добавляет элемент в конец списка
<code>void <b>push_front</b>(const Type&amp; val);</code> <code>void <b>push_front</b>(Type&amp;&amp; val);</code>	Добавляет элемент в начало списка
<code>const_reverse_iterator <b>rbegin</b>() const;</code> <code>reverse_iterator <b>rbegin</b>();</code>	Возвращает итератор, который обращается к первому элементу в перевернутом списке
<code>void <b>remove</b>(const Type&amp; val);</code>	Стирает элементы в списке, соответствующие заданному значению
<code>template &lt;class Predicate&gt;</code> <code>void <b>remove_if</b>(Predicate pred)</code>	Стирает элементы из списка, для которых выполняется заданный предикат
<code>const_reverse_iterator <b>rend</b>() const;</code> <code>reverse_iterator <b>rend</b>();</code>	Возвращает итератор, который обращается к местоположению, следующему за

	последним элементом в перевернутом списке
<code>void <b>resize</b>(size_type _Newsize);</code> <code>void <b>resize</b>(size_type _Newsize, Type val);</code>	Задаёт новый размер для списка
<code>void <b>reverse</b>();</code>	Изменяет порядок расположения элементов в списке в обратном порядке
<code>size_type <b>size</b>() const;</code>	Возвращает количество элементов в списке
<code>void <b>swap</b>(list&lt;Type, Allocator&gt;&amp; right);</code> <code>friend void <b>swap</b>(list&lt;Type, Allocator&gt;&amp; left, list&lt;Type, Allocator&gt;&amp; right)</code>	Меняет местами элементы двух списков
<code>void <b>merge</b>(list&lt;Type, Allocator&gt;&amp; right);</code> <code>template &lt;class Traits&gt;</code> <code>void <b>merge</b>(list&lt;Type, Allocator&gt;&amp; right, Traits comp);</code>	Удаляет элементы из списка аргументов, вставляет их в целевой список и упорядочивает новый комбинированный набор элементов в порядке возрастания или в каком-либо другом заданном порядке.
<code>void <b>sort</b>();</code> <code>template &lt;class Traits&gt;</code> <code>void <b>sort</b>(Traits comp);</code>	Упорядочивает элементы списка в порядке возрастания или по отношению к какому-либо другому заданному пользователем порядку
<code>// insert the entire source list</code> <code>void <b>splice</b>(const_iterator Where, list&lt;Type, Allocator&gt;&amp; Source);</code> <code>void <b>splice</b>(const_iterator Where, list&lt;Type, Allocator&gt;&amp;&amp; Source);</code> <code>// insert one element of the source list</code> <code>void <b>splice</b>(const_iterator Where, list&lt;Type, Allocator&gt;&amp; Source, const_iterator Iter);</code> <code>void <b>splice</b>(const_iterator Where, list&lt;Type, Allocator&gt;&amp;&amp; Source, const_iterator Iter);</code> <code>// insert a range of elements from the source list</code> <code>void <b>splice</b>(const_iterator Where, list&lt;Type, Allocator&gt;&amp; Source, const_iterator First, const_iterator Last);</code> <code>void <b>splice</b>(const_iterator Where, list&lt;Type, Allocator&gt;&amp;&amp; Source, const_iterator First, const_iterator Last);</code>	Удаляет элементы из исходного списка и вставляет их в список назначения
<code>void <b>unique</b>();</code> <code>template &lt;class BinaryPredicate&gt;</code> <code>void <b>unique</b>(BinaryPredicate pred);</code>	Удаляет из списка соседние повторяющиеся элементы или соседние элементы, удовлетворяющие какому-либо другому двоичному предикату

Чтобы достичь максимальной гибкости и переносимости для любого объекта, который подлежит хранению в списке, следует определить конструктор по умолчанию и оператор "<" (желательно определить и другие операторы сравнения). Более точные требования к объекту (как к потенциальному элементу списка) необходимо согласовывать в соответствии с документацией на используемый компилятор.

//Пример №6. Базовые операции контейнера `list`  
#include <iostream>



```

#include <list>
#include <string>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    list<string> lst; //создание пустого списка
    for (int i = 0; i < 10; i++)
        lst.push_back("Строка для списка " + to_string(i));
    cout << "Размер списка= " << lst.size() << endl;
    cout << "Содержимое списка: " << endl;
    list<string>::iterator listPtr = lst.begin();
    while (listPtr != lst.end()) {
        cout << *listPtr << endl;
        listPtr++;
    }
    return 0;
}

```

Результаты выполнения программы:

```

Размер списка= 10
Содержимое списка:
Строка для списка 0
Строка для списка 1
Строка для списка 2
Строка для списка 3
Строка для списка 4
Строка для списка 5
Строка для списка 6
Строка для списка 7
Строка для списка 8
Строка для списка 9

```

При выполнении эта программа создает пустой список строк. Затем в него помещается десять строк. Заполнение списка реализуется путем использования функции *push\_back()*, которая помещает каждое новое значение в конец существующего списка. После этого отображается размер списка и его содержимое. Содержимое списка выводится на экран в результате выполнения следующего кода:

```

list<string>::iterator listPtr = lst.begin();
while (listPtr != lst.end()) {
    cout << *listPtr << endl;
    listPtr++;
}

```

Здесь итератор *listPtr* инициализируется таким образом, чтобы он указывал на начало списка. При выполнении очередного прохода цикла итератор *listPtr* инкрементируется, чтобы указывать на следующий элемент списка. Этот цикл завершается, когда итератор *listPtr* указывает на конец списка. Применение подобных циклов — обычная практика при использовании библиотеки *STL*.

Поскольку списки являются двунаправленными, заполнение их элементами можно производить с обоих концов. Например, при выполнении следующей программы создается два списка, причем элементы одного из них расположены в порядке, обратном по отношению к другому.

```
//Пример №7. Добавление элементов в контейнер list
#include <iostream>
#include <list>
#include <string>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    list<string> lst;
    list<string> reverseLst;
    int i;
    for (i = 0; i < 10; i++) lst.push_back("Строка для списка " +
to_string(i));
    cout << "Размер списка lst = " << lst.size() << endl;
    cout << "Исходное содержимое списка: " << endl;
    list<string>::iterator listPtr;
    //Удаляем элементы из списка lst и помещаем их в список
reverseLst в обратном порядке.
    while (!lst.empty()) {
        listPtr = lst.begin();
        cout << *listPtr << endl;
        reverseLst.push_front(*listPtr);
        lst.pop_front();
    }
    cout << endl << endl;
    cout << "Размер списка reverseLst = ";
    cout << reverseLst.size() << endl;
    cout << "Реверсированное содержимое списка: " << endl;
    listPtr = reverseLst.begin();
    while (listPtr != reverseLst.end()) {
        cout << *listPtr << endl;
        listPtr++;
    }
    return 0;
}
```

Результаты работы программы:

```

Размер списка lst = 10
Исходное содержимое списка:
Строка для списка 0
Строка для списка 1
Строка для списка 2
Строка для списка 3
Строка для списка 4
Строка для списка 5
Строка для списка 6
Строка для списка 7
Строка для списка 8
Строка для списка 9

Размер списка reverseLst = 10
Реверсированное содержимое списка:
Строка для списка 9
Строка для списка 8
Строка для списка 7
Строка для списка 6
Строка для списка 5
Строка для списка 4
Строка для списка 3
Строка для списка 2
Строка для списка 1
Строка для списка 0

```

В этой программе список реверсируется путем удаления элементов с начала списка *lst* и занесения их в начало списка *reverseLst*.

## СОРТИРОВКА ЭЛЕМЕНТОВ СПИСКА

Список можно отсортировать с помощью метода-элемента *sort()*. При выполнении следующей программы создается список случайно выбранных целых чисел, который затем упорядочивается по возрастанию.

```

//Пример №8. Сортировка содержимого контейнера list
#include <iostream>
#include <list>
#include <cstdlib>
#include <time.h>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    srand(time(0));
    list<int> lst;
    int i;
    //Создание списка случайно выбранных целых чисел
    for (i = 0; i < 10; i++) lst.push_back(rand());
    cout << "Исходное содержимое списка:\n";
    list<int>::iterator p = lst.begin();
    while (p != lst.end()) {
        cout << *p << " ";
        p++;
    }
    cout << endl << endl;
    //Сортировка списка
    lst.sort();
    cout << "Отсортированное содержимое списка:\n";
    p = lst.begin();
    while (p != lst.end()) {
        cout << *p << " ";
        p++;
    }
    return 0;
}

```

Результат работы программы:

```
Исходное содержимое списка:
8970 23690 15185 23510 26543 20509 13907 6877 1709 6690

Отсортированное содержимое списка:
1709 6690 6877 8970 13907 15185 20509 23510 23690 26543
```

## ОБЪЕДИНЕНИЕ ОДНОГО СПИСКА С ДРУГИМ

Один упорядоченный список можно объединить с другим. В результате получится упорядоченный список, который включает содержимое двух исходных списков. Новый список остается в вызывающем списке, а второй список становится пустым.

В следующем примере выполняется слияние двух списков. Первый список содержит буквы *ACEGI*, а второй— буквы *BDFHJ*. Эти списки затем объединяются, в результате чего образуется упорядоченная последовательность букв *ABCDEFGHIIJ*.

```
//Пример №9. Слияние двух списков
#include <iostream>
#include <list>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    list<char> lst1, lst2;
    int i;
    for (i = 0; i < 10; i += 2) lst1.push_back('A' + i);
    for (i = 1; i < 11; i += 2) lst2.push_back('A' + i);
    cout << "Содержимое списка lst1: ";
    list<char>::iterator p = lst1.begin();
    while (p != lst1.end()) {
        cout << *p;
        p++;
    }
    cout << endl << endl;
    cout << "Содержимое списка lst2: ";
    p = lst2.begin();
    while (p != lst2.end()) {
        cout << *p;
        p++;
    }
    cout << endl << endl;
    //объединение и упорядочивание содержимого двух списков
    lst1.merge(lst2);
    if (lst2.empty())
        cout << "Список lst2 теперь пуст.\n";
    cout << "Содержимое списка lst1 после объединения:\n";
    p = lst1.begin();
    while (p != lst1.end()) {
        cout << *p;
        p++;
    }
    return 0; }
```

Результаты выполнения этой программы:

```
Содержимое списка lst1: ACEGI
Содержимое списка lst2: BDFHJ
Список lst2 теперь пуст.
Содержимое списка lst1 после объединения:
ABCDEFGHIJ
```

## ХРАНЕНИЕ В СПИСКЕ ОБЪЕКТОВ КЛАССОВ

Рассмотрим пример, в котором список используется для хранения объектов типа *Student*. Обратите внимание на то, что для объектов типа *Student* перегружены операторы "<", ">", "!=" и "==". Для некоторых компиляторов может оказаться излишним определение всех этих операторов или же придется добавить некоторые другие. В библиотеке *STL* эти функции используются для определения упорядочения и равенства объектов в контейнере. Несмотря на то, что список не является упорядоченным контейнером, необходимо иметь средство сравнения элементов, которое применяется при их поиске, сортировке или объединении.

//Пример №10. Хранение в списке объектов класса

```
#include <iostream>
#include <list>
#include <cstring>
#include <iomanip>
using namespace std;
class Student {
    float mark;
public:
    Student() { mark = 0; }
    Student(float mark) {
        this->mark = mark;
    }
    float getMark() { return mark; }
    friend bool operator<(const Student& o1, const Student& o2);
    friend bool operator>(const Student& o1, const Student& o2);
    friend bool operator==(const Student& o1, const Student& o2);
    friend bool operator!=(const Student& o1, const Student& o2);
};
bool operator<(const Student& o1, const Student& o2) {
    return o1.mark < o2.mark;
}
bool operator>(const Student& o1, const Student& o2) {
    return o1.mark > o2.mark;
}
bool operator==(const Student& o1, const Student& o2) {
    return o1.mark == o2.mark;
}
bool operator!=(const Student& o1, const Student& o2) {
    return o1.mark != o2.mark;
}
int main() {
```



```

system("chcp 1251");
system("cls");
float i;
//Создание первого списка
list<Student> group1;
for (i = 0; i < 7; i += 0.1) group1.push_back(Student(i));
cout << "Оценки первой группы: " << endl;
list<Student>::iterator p = group1.begin();
while (p != group1.end()) {
    cout << fixed << setprecision(2) << p->getMark() << " ";
    p++;
}
cout << endl;
//Создание второго списка.
list<Student> group2;
for (i = 0; i < 8; i += 0.2) group2.push_back(Student(i));
cout << "Оценки второй группы: " << endl;
p = group2.begin();
while (p != group2.end()) {
    cout << p->getMark() << " ";
    p++;
}
cout << endl;
//объединяем списки lst1 и lst2
group1.merge(group2);
//Отображаем объединенный список
cout << "Объединенный набор оценок двух групп: " << endl;
p = group1.begin();
while (p != group1.end()) {
    cout << p->getMark() << " ";
    p++;
}
return 0;
}

```

Программа создает два списка объектов типа *Student* и отображает их содержимое. Затем выполняется объединение этих двух списков с последующим отображением нового содержимого результирующего списка. Результаты работы программы:

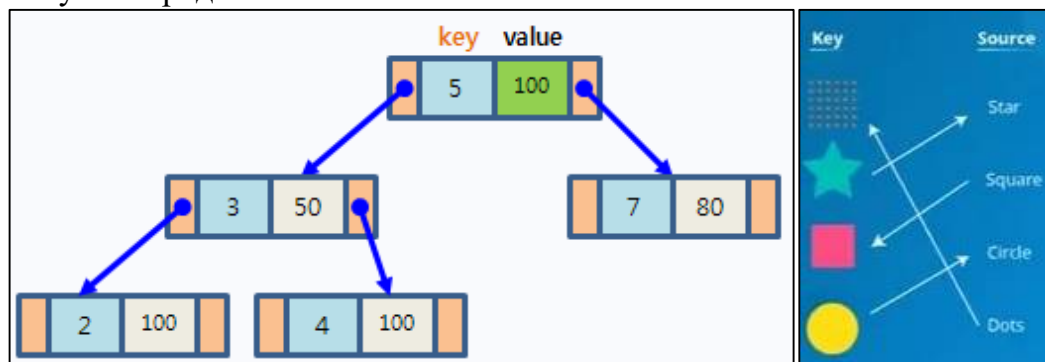
```

Оценки первой группы:
0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.00 1.10 1.20 1.30 1.40 1.50 1.60 1.70 1.80 1.90 2.00 2.10 2.20 2.30 2.40 2.50 2.60 2.70 2.80 2.90 3.00 3.10 3.20 3.30 3.40 3.50 3.60 3.70 3.80 3.90 4.00 4.10 4.20 4.30 4.40 4.50 4.60 4.70 4.80 4.90 5.00 5.10 5.20 5.30 5.40 5.50 5.60 5.70 5.80 5.90 6.00 6.10 6.20 6.30 6.40 6.50 6.60 6.70 6.80 6.90 7.00
Оценки второй группы:
0.00 0.20 0.40 0.60 0.80 1.00 1.20 1.40 1.60 1.80 2.00 2.20 2.40 2.60 2.80 3.00 3.20 3.40 3.60 3.80 4.00 4.20 4.40 4.60 4.80 5.00 5.20 5.40 5.60 5.80 6.00 6.20 6.40 6.60 6.80 7.00 7.20 7.40 7.60 7.80 8.00
Объединенный набор оценок двух групп:
0.00 0.00 0.10 0.20 0.20 0.30 0.40 0.40 0.50 0.60 0.60 0.70 0.80 0.80 0.90 1.00 1.00 1.10 1.20 1.20 1.30 1.40 1.40 1.50 1.60 1.60 1.70 1.80 1.80 1.90 2.00 2.00 2.10 2.20 2.20 2.30 2.40 2.40 2.50 2.60 2.60 2.70 2.80 2.80 2.90 3.00 3.00 3.10 3.20 3.20 3.30 3.40 3.40 3.50 3.60 3.60 3.70 3.80 3.80 3.90 4.00 4.00 4.10 4.20 4.20 4.30 4.40 4.40 4.50 4.60 4.60 4.70 4.80 4.80 4.90 5.00 5.00 5.10 5.20 5.20 5.30 5.40 5.40 5.50 5.60 5.60 5.70 5.80 5.80 5.90 6.00 6.00 6.10 6.20 6.20 6.30 6.40 6.40 6.50 6.60 6.60 6.70 6.80 6.80 6.90 7.00 7.00 7.20 7.40 7.60 7.80 8.00
0

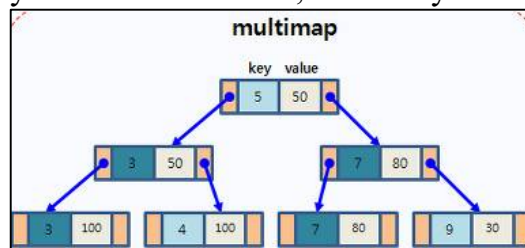
```

## КОНТЕЙНЕР MAP

Отображение, таблица (*map*) — это ассоциативный контейнер. Класс *map* поддерживает ассоциативный контейнер, в котором уникальным ключам соответствуют определенные значения.



Ключ — это имя, которое присвоено некоторому значению. После того как значение сохранено в контейнере, к нему можно получить доступ, используя его ключ. Таким образом, в самом широком смысле отображение — это список пар "ключ-значение". Если известен ключ, то можно легко найти значение. Например, можно определить отображение, в котором в качестве ключа используется имя человека, а в качестве значения — его телефонный номер. ***map* может хранить только уникальные ключи.** Ключи-дубликаты не разрешены. Чтобы создать отображение, которое бы позволяло хранить неуникальные ключи, используется класс *multimap*.



Контейнер *map* имеет следующую шаблонную спецификацию.

```
template <class Key, class Type, class Traits = less<Key>, class
Allocator=allocator<pair <const Key, Type>>>
class map;
```

Где параметр *Key* определяет ключевой тип данных, который будет храниться в отображении, параметр *Type* определяет тип данных элемента, который будет храниться в отображении, параметр *Traits* определяет тип, предоставляющий объект функции, который может сравнивать два значения элементов в качестве ключей сортировки для определения их относительного порядка в отображении карте. Этот аргумент является необязательным, и двоичный предикат *less<Key>* является значением по умолчанию. Параметр *Allocator* определяет тип, представляющий объект распределителя, который инкапсулирует сведения о выделении и освобождении памяти карты. Этот аргумент является необязательным, и значение по умолчанию — *allocator<pair<const Key, Type>>*.

Класс *map* имеет следующие конструкторы.

```
map();
explicit map(const Traits& Comp);
map(const Traits& Comp, const Allocator& Al);
map(const map& Right);
map(map&& Right);
map(initializer_list<value_type> IList);
map(initializer_list<value_type> IList, const Traits& Comp);
map(initializer_list<value_type> IList, const Traits& Comp,
const Allocator& Allocator);
template <class InputIterator>
map(InputIterator First, InputIterator Last);
template <class InputIterator>
map(InputIterator First, InputIterator Last, const Traits&
Comp);
template <class InputIterator>
map(InputIterator First, InputIterator Last, const Traits&
Comp, const Allocator& Al);
```

Где параметр *Al* определяет класс распределителя хранилища, используемый для этого отображения, параметр *Comp* определяет функцию сравнения признаков типа *const Traits*, используемая для упорядочения элементов в отображении карте, по умолчанию имеет значение *hash\_compare*. Параметр *Right* определяет отображение, копией которой должен быть сконструированный набор, параметр *First* определяет положение первого элемента в диапазоне копируемых элементов, параметр *Last* определяет положение первого элемента за пределами диапазона копируемых элементов, параметр *IList* определяет список инициализации, из которого должны быть скопированы элементы.

В общем случае любой объект, используемый в качестве ключа, должен определять конструктор по умолчанию и перегружать оператор "<" (а также другие необходимые операторы сравнения).

Методы, определенные для класса *map*, представлены в таблице №7.

Таблица №7 – Методы класса *map*

Сигнаutra метода	Назначение метода
<i>const_iterator</i> <b>begin</b> () <i>const</i> ; <i>iterator</i> <b>begin</b> ();	Двунаправленный итератор, указывающий на первый элемент отображения
<i>void</i> <b>clear</b> ();	Удаляет все элементы отображения
<i>size_type</i> <b>count</b> ( <i>const</i> <i>Key</i> & <i>key</i> ) <i>const</i> ;	Возвращает количество элементов в отображении, ключ которых соответствует ключу, заданному параметром
<i>bool</i> <b>empty</b> () <i>const</i> ;	Проверяет, пустое ли отображение
<i>const_iterator</i> <b>end</b> () <i>const</i> ; <i>iterator</i> <b>end</b> ();	Возвращает итератор на конец отображения
<i>pair</i> < <i>const_iterator</i> , <i>const_iterator</i> > <b>equal_range</b> ( <i>const</i> <i>Key</i> & <i>key</i> ) <i>const</i> ; <i>pair</i> < <i>iterator</i> , <i>iterator</i> > <b>equal_range</b> ( <i>const</i> <i>Key</i> & <i>key</i> );	Возвращает пару итераторов, представляющих нижнюю границу ключа и верхнюю границу ключа
<i>iterator</i> <b>erase</b> ( <i>const_iterator</i> <i>Where</i> ); <i>iterator</i> <b>erase</b> ( <i>const_iterator</i> <i>First</i> , <i>const_iterator</i> <i>Last</i> ); <i>size_type</i> <b>erase</b> ( <i>const</i> <i>key_type</i> & <i>Key</i> );	Удаляет элемент или диапазон элементов отображения из заданных позиций или удаляет элементы, соответствующие указанному ключу
<i>iterator</i> <b>find</b> ( <i>const</i> <i>Key</i> & <i>key</i> ); <i>const_iterator</i> <b>find</b> ( <i>const</i> <i>Key</i> & <i>key</i> ) <i>const</i> ;	Возвращает итератор, который ссылается на расположение элемента в отображении, имеющего ключ, эквивалентный указанному ключу
<i>allocator_type</i> <b>get_allocator</b> () <i>const</i> ;	Возвращает копию объекта распределителя, используемого для построения отображения
<i>pair</i> < <i>iterator</i> , <i>bool</i> > <b>insert</b> ( <i>const</i> <i>value_type</i> & <i>Val</i> ); <i>template</i> < <i>class</i> <i>ValTy</i> > <i>pair</i> < <i>iterator</i> , <i>bool</i> > <b>insert</b> ( <i>ValTy</i> && <i>Val</i> ); <i>iterator</i> <b>insert</b> ( <i>const_iterator</i> <i>Where</i> , <i>const</i> <i>value_type</i> & <i>Val</i> ); <i>template</i> < <i>class</i> <i>ValTy</i> > <i>iterator</i> <b>insert</b> ( <i>const_iterator</i> <i>Where</i> , <i>ValTy</i> && <i>Val</i> ); <i>template</i> < <i>class</i> <i>InputIterator</i> > <i>void</i> <b>insert</b> ( <i>InputIterator</i> <i>First</i> , <i>InputIterator</i> <i>Last</i> ); <i>void</i> <b>insert</b> ( <i>initializer_list</i> < <i>value_type</i> > <i>IList</i> );	Добавляет элемент или диапазон элементов в отображение
<i>key_compare</i> <b>key_comp</b> () <i>const</i> ;	Извлекает копию объекта сравнения, используемого для упорядочения ключей в отображении
<i>iterator</i> <b>lower_bound</b> ( <i>const</i> <i>Key</i> & <i>key</i> ); <i>const_iterator</i> <b>lower_bound</b> ( <i>const</i> <i>Key</i> & <i>key</i> ) <i>const</i> ;	Возвращает итератор на первый элемент отображения со значением ключа, равным или превышающим значение указанного ключа

<code>size_type max_size() const;</code>	Возвращает максимальную длину отображения
<code>Type&amp; <b>operator</b>[(const Key&amp; key); Type&amp; <b>operator</b>[(Key&amp;&amp; key);</code>	Возвращает ссылку на элемент, заданный параметром <i>key</i> . Если этого элемента не существует, то вставляет элемент в отображение с заданным значением ключа
<code>const_reverse_iterator <b>rbegin</b>() const; reverse_iterator <b>rbegin</b>();</code>	Возвращает итератор, адресующий первый элемент в перевернутом отображении
<code>const_reverse_iterator <b>rend</b>() const; reverse_iterator <b>rend</b>();</code>	Возвращает итератор, указывающий за последним элементом в перевернутом отображении
<code>size_type size() const;</code>	Возвращает количество элементов в отображении
<code>void swap(map&lt;Key, Type, Traits, Allocator&gt;&amp; right);</code>	Обменивается элементами двух отображений
<code>iterator <b>upper_bound</b>(const Key&amp; key); const_iterator <b>upper_bound</b>(const Key&amp; key) const;</code>	Возвращает итератор на первый элемент отображения, у которого ключ имеет значение, превышающее значение указанного ключа
<code>value_compare <b>value_comp</b>() const;</code>	Возвращает объект функции, который определяет порядок элементов в отображении

Пары "ключ-значение" хранятся в отображении как объекты класса *pair*, который имеет следующую шаблонную спецификацию.

```
struct pair{
    typedef T1 first_type; //тип ключа
    typedef T2 second_type; //тип значения
    T1 first; //содержит ключ
    T2 second; //содержит значение
    //Конструкторы
    constexpr pair();
    pair(const pair&) = default;
    pair(pair&&) = default;
    constexpr pair(const T1& Val1, const T2& Val2);
    template <class Other1, class Other2>
    constexpr pair(const pair<Other1, Other2>& Right);
    template <class Other1, class Other2>
    constexpr pair(const pair <Other1 Val1, Other2 Val2>&&
    Right);
    template <class Other1, class Other2>
    constexpr pair(Other1&& Val1, Other2&& Val2);
    template <class... Args1, class... Args2>
    pair(piecewise_construct_t, tuple<Args1...> first_args,
    tuple<Args2...> second_args);
    pair& operator=(const pair& p);
    template<class U1, class U2> pair& operator=(const pair<U1,
    U2>& p);
    pair& operator=(pair&& p) noexcept(see below );
    template<class U1, class U2> pair& operator=(pair<U1, U2>&&
    p);
    void swap(pair& p) noexcept(see below );
};
```

Элемент *first* содержит ключ, а элемент *second* — значение, соответствующее этому ключу.



Создать пару "ключ-значение" можно либо с помощью конструкторов класса *pair*, либо путем вызова функции *make\_pair()*, которая создает парный объект на основе типов данных, используемых в качестве параметров. Функция *make\_pair()* — обобщенная функция, прототип которой имеет следующие варианты перегрузки:

```
template <class T, class U>
    pair<T, U> make_pair(T& Val1, U& Val2);
template <class T, class U>
    pair<T, U> make_pair(T& Val1, U&& Val2);
template <class T, class U>
    pair<T, U> make_pair(T&& Val1, U& Val2);
template <class T, class U>
    pair<T, U> make_pair(T&& Val1, U&& Val2);
```

Функция *make\_pair()* возвращает парный объект, состоящий из значений, типы которых заданы параметрами *T* и *U*. **Преимущество использования функции *make\_pair()* состоит в том, что типы объектов, объединяемых в пару, определяются автоматически компилятором, а не явно задаются программистом.**

Возможности использования отображения демонстрируется в следующем примере. В примере в отображении сохраняется 10 пар "ключ-значение". Ключом служит строка, а значением — целое число. После сохранения пар в отображении пользователю предлагается ввести ключ, после чего выводится значение, связанное с этим ключом.

```
//Пример №11. Использование отображения
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    map<string, float> group;
    //Помещаем пары в отображение
    group.insert(pair<string, float>("Воскресенская Наталья", 9.0));
    group.insert(pair<string, float>("Московская Елизавета", 5.0));
    group.insert(pair<string, float>("Ларченко Иван", 7.5));
    group.insert(pair<string, float>("Ларченко Степан", 7.6));
    group.insert(pair<string, float>("Сидоров Геннадий", 6.5));
    group.insert(pair<string, float>("Переходцева Ирина", 7.5));
    string name;
    char nameChar[256];
    map<string, float>::iterator mapPtr;
    while (true) {
        cout << "Введите ключ: ";
        cin.getline(nameChar, 255);
        //cout << nameChar;
```

```

        name += nameChar;
        //Находим значение по заданному ключу
        mapPtr = group.find(name);
        if (mapPtr != group.end())
            cout << "Оценка студента равна " << mapPtr->second <<
endl;

        else cout << "Такого студента нет в группе\n";
        name.erase();
    }
    return 0;
}

```

Результат работы программы:

```

Введите ключ: Переходцева Ирина
Оценка студента равна 7.5
Введите ключ: Воскресенская Наталья
Оценка студента равна 9
Введите ключ: Сидоров Геннадий
Оценка студента равна 6.5
Введите ключ: Пригожин Иван
Такого студента нет в группе
Введите ключ: _

```

Обратите внимание на использование шаблонного класса *pair* для построения пар "ключ-значение". Типы данных, задаваемые *pair*-выражением, должны соответствовать типам отображения, в которое вставляются эти пары.

После инициализации отображения ключами и значениями можно выполнять поиск значения по заданному ключу, используя функцию *find()*. Эта функция возвращает итератор, который указывает на нужный элемент или на конец отображения, если заданный ключ не был найден. При обнаружении совпадения значение, связанное с ключом, можно найти в элементе *second* парного объекта типа *pair*.

В примере пары "ключ-значение" создавались явным образом с помощью шаблона *pair<string, float>*. И хотя в этом нет ничего неправильного, зачастую проще использовать с этой целью функцию *make\_pair()*, которая создает *pair*-объект на основе типов данных, используемых в качестве параметров. Например, эта строка кода также позволит вставить в отображение *group* пары "ключ-значение" (при использовании предыдущей программы):

```
group.insert(make_pair("Воскресенская Наталья", 9.0));
```

## ХРАНИЛИЩЕ В ОТОБРАЖЕНИИ ОБЪЕКТОВ КЛАССА

Подобно другим контейнерам, отображение можно использовать для хранения объектов создаваемых пользователем типов. Например, следующая программа создает простой словарь на основе отображения слов с их значениями. Но сначала она создает два класса *Word* и *Meaning*. Поскольку отображение поддерживает отсортированный список ключей, программа также определяет для объектов типа *Word* оператор "<". В общем случае

оператор "<" следует определять для любых классов, которые необходимо использовать в качестве ключей.

```
//Пример №12. Использование отображения для создания словаря
#include <iostream>
#include <map>
#include <string>
using namespace std;
class Word {
    char description[80];
public:
    Word() { strcpy_s(description, ""); }
    Word(const char* description) { strcpy_s(this->description,
description); }
    char* getDescription() { return description; }
};
bool operator<(Word w1, Word w2) {
    return strcmp(w1.getDescription(), w2.getDescription()) < 0;
}
class Meaning {
    char description[150];
public:
    Meaning() { strcpy_s(description, ""); }
    Meaning(const char* description) { strcpy_s(this->description,
description); }
    char* getDescription() { return description; }
};
int main() {
    system("chcp 1251");
    system("cls");
    map<Word, Meaning> dictionary;
    //Помещаем в отображение объекты классов word и meaning
    dictionary.insert(pair<Word, Meaning>(Word("апгрейд"),
Meaning("Обновление/модернизация аппаратного обеспечения (железа)")));
    dictionary.insert(pair<Word, Meaning>(Word("апдейт"),
Meaning("обновление/модернизация программного обеспечения (софта)")));
    dictionary.insert(pair<Word, Meaning>(Word("апрув"),
Meaning("подтверждение, согласие, одобрение чего-либо")));
    dictionary.insert(pair<Word, Meaning>(Word("аутсорс"),
Meaning("передача предприятием/компанией выполнение определенной
работы специалистам вне штата"))));
    //По заданному слову находим его значение
    char description[80];
    while (true) {
        cout << "Введите слово: ";
        cin.getline(description, 255);
        map<Word, Meaning>::iterator mapPtr;
        mapPtr = dictionary.find(Word(description));
        if (mapPtr != dictionary.end())
            cout << "Определение: " << mapPtr-
>second.getDescription() << endl;
```

```

        else cout << "Такого слова в словаре нет.\n";
    }
    return 0;
}

```

Результат работы программы:

```

Введите слово: апгрейд
Определение: Обновление/модернизация аппаратного обеспечения (
железа)
Введите слово: апдейт
Определение: обновление/модернизация программного обеспечения
(софта)
Введите слово: футсорс
Такого слова в словаре нет.
Введите слово: аутсорс
Определение: передача предприятием/компанией выполнение опреде
ленной работы специалистам вне штата
Введите слово:

```

В этой программе каждый элемент отображения представляет собой символьный массив, который содержит строку с завершающим нулем.

### **КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:**

1. Что такое контейнер в библиотеке *STL*?
2. Что такое итератор в библиотеке *STL*?
3. Что такое алгоритм в библиотеке *STL*?
4. Какие виды итераторов существуют в библиотеке *STL*?
5. Что представляет собой предикат в библиотеке *STL*?
6. Что собой представляет контейнер вектор (*vector*) в *C++*?
7. Что собой представляет контейнер список (*list*) в *C++*?
8. Что собой представляет контейнер отображение (*map*) в *C++*?
9. В чем различие между контейнерами *list*, *vector*, *map*?
10. Что собой представляют ассоциативные и последовательные контейнеры? В чем между ними различие?

### **ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:**

- 1.1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.
2. Ответить на контрольные вопросы лабораторной работы.
3. Разработать алгоритм программы по индивидуальному заданию.
4. Написать, отладить и проверить корректность работы созданной программы.
5. Написать электронный отчет по выполненной лабораторной работе.

**Отчет должен быть оформлен по стандарту БГУИР ([Стандарт предприятия СТП 01-2017 "Дипломные проекты \(работы\). Общие требования"](#)) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания

5. весь код решения индивидуального задания, разбитый на необходимые типы файлов

6. скриншоты выполнения индивидуального задания

7. выводы по лабораторной работе

**В РАМКАХ ВСЕГО КУРСА «ООП» ВСЕ ЛАБОРАТОРНЫЕ РАБОТЫ НА ЯЗЫКЕ C++ ДОЛЖНЫ ХРАНИТЬСЯ В ОДНОМ РЕШЕНИИ (SOLUTION), В КОТОРОМ ДОЛЖНЫ БЫТЬ СОЗДАНЫ ОТДЕЛЬНЫЕ ПРОЕКТЫ (PROJECTS) ДЛЯ КАЖДОЙ ЛАБОРАТОРНОЙ РАБОТЫ. ВО ВСЕХ ПРОЕКТАХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. ВСЕ РЕШАЕМЫЕ ЗАДАЧИ ДОЛЖНЫ БЫТЬ РЕАЛИЗОВАНЫ, ИСПОЛЬЗУЯ НЕОБХОДИМЫЕ КЛАССЫ И ОБЪЕКТЫ.**

### **ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ №5:**

В каждом индивидуальном задании необходимо создать контейнеры, которые будут хранить объекты классов по предметной области, указанной в таблице (класс должен содержать функционал по предметной области). Для контейнера реализовать добавление, удаление, редактирование, вывод содержимого контейнера на экран и в файл, поиск и сортировку элементов. Необходимо создать удобное пользовательское меню.

№	Контейнер	Тема
1.	<i>list, map</i>	студенты ВУЗа
2.	<i>vector, map</i>	банковские сотрудники
3.	<i>map, list</i>	каталог книг
4.	<i>vector, map</i>	тестирование знаний студентов
5.	<i>list, map</i>	транспортная техника
6.	<i>vector, map</i>	студенты ВУЗа
7.	<i>map, vector</i>	медицинские работники
8.	<i>vector, map</i>	каталог книг
9.	<i>list, map</i>	банковские операции
10.	<i>vector, map</i>	строительная техника
11.	<i>map, list</i>	медицинские работники
12.	<i>vector, map</i>	банковские сотрудники
13.	<i>list, map</i>	печатная продукция
14.	<i>vector, map</i>	бронирование авиабилетов
15.	<i>map, list</i>	продажа и покупка недвижимости
16.	<i>vector, map</i>	медицинские работники
17.	<i>list, map</i>	банковские сотрудники
18.	<i>vector, map</i>	ассортимент услуг
19.	<i>map, list</i>	бронирование авиабилетов
20.	<i>vector, map</i>	бытовая техника
21.	<i>list, map</i>	участники спортивных мероприятий
22.	<i>vector, map</i>	банковские сотрудники
23.	<i>list, map</i>	издательство печатной продукции

24.	<i>vector, map</i>	документооборот
25.	<i>map, list</i>	сотрудники ИТ-организации
26.	<i>vector, map</i>	розничная продажа товаров и услуг
27.	<i>list, map</i>	тестирование по английскому языку
28.	<i>map, list</i>	работники строительной организации
29.	<i>vector, map</i>	сотрудники библиотеки
30.	<i>list, map</i>	сельскохозяйственная продукция