БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
факультет радиофизики и компьютерных технологий
кафедра информатики и компьютерных систем

Н.В. Серикова

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

к лабораторному практикуму

БИБЛИОТЕКА СТАНДАРТНЫХ ШАБЛОНОВ STL

по курсу
«ПРОГРАММИРОВАНИЕ»

2020
МИНСК

Практическое руководство к лабораторному практикуму «КЛАССЫ» по курсу «ПРОГРАММИРОВАНИЕ» предназначено для студентов, изучающих базовый курс программирования на языке C++, специальностей «Радиофизика», «Физическая электроника», «Компьютерная безопасность».

Руководство содержит некоторый справочный материал, примеры решения типовых задач с комментариями.

Все примеры протестированы в среде Microsoft Visual Studio 2005.

Автор будет признателен всем, кто поделится своими соображениями по совершенствованию данного пособия.

Возможные предложения и замечания можно присылать по адресу:

E-mail: *Serikova@bsu.by*

ОГЛАВЛЕНИЕ

БИБЛИОТЕКА СТАНДАРТНЫХ ШАБЛОНОВ STL

Библиотека стандартных шаблонов C++ (**Standart Template Library**) обеспечивает стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных.

В частности, в библиотеке STL поддерживаются **вектора (vector), списки (list), очереди (queue), стеки (stack).** Определены процедуры доступа к этим структурам данных.

Ядро библиотеки образуют три элемента: **контейнеры, алгоритмы и итераторы**.

Контейнеры – объекты, предназначенные для хранения других объектов. Например, в класса *vector* определяется динамический массив, в классе *queue* – очередь, в классе *list* – линейный список. В каждом классе-контейнере определяется набор функций для работы с этим контейнером. Например, список содержит функции для вставки, удаления, слияния элементов. В стеке – функции для размещения элемента в стек и извлечения элемента из стеа.

Алгоритмы выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнера.

Итераторы – объекты, которые к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива. С итераторами можно работать так же как с указателями.

Существуют 5 типов итераторов.

Итератор	**Описание**
Произвольного доступа	Используется для считывания и записи значений. Доступ к элементам произвольный
Двунаправленный	Используется для считывания и записи значений. Может проходить контейнер в обоих наравлениях
Однонаправленный	Используется для считывания и записи значений. Может проходить контейнер только в одном наравлении
Ввода	Используется только для считывания значений. Может проходить контейнер только в одном наравлении
Вывода	Используется только для записи значений. Может проходить контейнер только в одном наравлении

Классы-контейнеры, определенные в STL

Контейнер	Описание	Заголовок
bitset	Множество битов	<bitset>
deque	Двустороняя очередь	<deque>
list	Линейный список	<list>
map	Ассоциативный список для хранения пар (ключ/ значение), где с каждым ключом связано одно значение	<map>
multimap	Ассоциативный список для хранения пар (ключ/ значение), где с каждым ключом связано два или более значений	<map>
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>
priority-queue	Очередь с приоритетом	<queue>
queue	Очередь	<queue>
set	Множество, в котором каждый элемент уникален	<set>
stack	Стек	<stack>
string	Строка символов	<string>
vector	Динамический массив	<vector>

Имена типов элементов, конкретизированных с помощью ключевого слова typedef, входящих в объявление классов-шаблонов:

Согласованное имя типа	Описание
size_type	Интегральный тип, эквивалентный типу size_t
reference	Ссылка на элемент
const_reference	Постоянная ссылка на элемент
iterator	Итератор
const_iterator	Постоянный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Постоянный обратный итератор
value_type	Тип хранящегося в контейнере значения
allocator_type	Тип распределителя памяти
key_type	Тип ключа
key_compare	Тип функции, которая сравнивает два ключа
value_compare	Тип функции, которая сравнивает два значения

ВЕКТОРЫ

Шаблон для класса vector:

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

Ключевое слово *Allocator* задает распределитель памяти, который по умолчанию является стандартным.

Определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());

explicit vector(size_type число, const T &значение = T(),
                const Allocator &a = Allocator());

vector(const vector<T,Allocator>&объект);

template <class InIter>vector(InIter начало, InIter конец,
                        const Allocator &a = Allocator());
```

Определены операторы сравнения:
`== < <= != > >=`

Определен оператор **[]**

Функции-члены класса *vector*

Функция-член	Описание
template<class lnlter> void assign (lnlter начало, Inlter конец);	Присваивает вектору последовательность, определенную итераторами *начало* и *конец*
template<class Size, class T> void assign (Size число, const T &значение = T());	Присваивает вектору *число* элементов, причем значение каждого элемента равно параметру *значение*
reference at(slze_type i); **const_reference at(size_type i) const;**	Возвращает ссылку на элемент, заданный параметром i
reference back(); **const_reference back() const;**	Возвращает ссылку на последний элемент вектора
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента вектора
size_type capacity() const;	Возвращает текущую емкость вектора, т. е. то число элементов, которое можно разместить в векторе без необходимости выделения дополнительной области памяти
void clear();	Удаляет все элементы вектора
bool empty() const;	Возвращает истину, если вызывающий вектор пуст, в противном случае возвращает ложь
iterator end(); **const_iterator end() const;**	Возвращает итератор конца вектора
iterator erase(iterator i);	Удаляет элемент, на который указывает итератор *i*. Возвращает итератор элемента, который расположен следующим за удаленным
iterator erase (iterator начало, iterator конец);	Удаляет элементы, заданные между итераторами *начало* и *конец.* Возвращает итератор элемента, который расположен следующим за последним удаленным
reference front(); **const_reference front() const;**	Возвращает ссылку на первый элемент вектора.
allocator_type get_allocator() const;	Возвращает распределитель памяти вектора
iterator insert(iterator i, const T &значение = T());	Вставляет параметр *значение* перед элементом, заданным итератором *i*. Возвращает итератор элемента

Функция-член	Описание
void insert(iterator i, size_type число, const T &значение);	Вставляет *число* копий параметра *значение* перед элементом, заданным итератором *i*
template<class InIter> void insert(iterator i, InIter начало, InIter конец);	Вставляет последовательность, определенную между итераторами *начало* и *конец,* перед элементом, заданным итератором *i*
size_type max_size() const;	Возвращает максимальное число элементов, которое может храниться в векторе
reference operator[] (size_type i) const; **const_reference operator[](size_type) const;**	Возвращает ссылку на элемент, заданный параметром *i*
void pop_back();	Удаляет последний элемент вектора
void push_back(const T &значение);	Добавляет в конец вектора элемент, значение которого равно параметру значение
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца вектора
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала вектора
void reserve(size_type число);	Устанавливает емкость вектора равной, по меньшей мере, параметру *число* элементов
void resize (size_type число, T значение = T();	Изменяет размер вектора в соответствии с параметром *число.* Если при этом вектор удлиняется, то добавляемые в конец вектора элементы получают значение, заданное параметром *значение*
size_type size() const;	Возвращает хранящееся на данный момент в векторе число элементов
void swap(vector<T, Allocator> &объект);	Обменивает элементы, хранящиеся в вызывающем векторе, с элементами в объекте *объект*

СПИСКИ

Шаблон для класса list:

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

Ключевое слово *Allocator* задает распределитель памяти, который по умолчанию является стандартным.

Определены следующие конструкторы:

```
explicit list(const Allocator &a = Allocator());

explicit list(size_type число, const T &значение = T(),
                const Allocator &a = Allocator());

list(const list<T,Allocator>&объект);

template <class InIter>list(InIter начало, InIter конец,
                        const Allocator &a = Allocator());
```

Определены операторы сравнения:
`== < <= != > >=`

Функции-члены класса *list*

Функция-член	**Описание**
template<class lnIter> void assign(lnIter *начало*,InIter *конец);	Присваивает списку последовательность, определенную итераторами *начало* и *конец*
template<class Size, class T> void assign (Size *число*,const T &*значение* = T());	Присваивает списку *число* элементов, причем значение каждого элемента равно параметру *значение*
reference back(); **const_reference back() const;**	Возвращает ссылку на последний элемент списка
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента списка
void clear();	Удаляет все элементы списка
bool empty() const;	Возвращает истину, если вызывающий список пуст, в противном случае возвращает ложь
iterator end(); **const_iterator end() const;**	Возвращает итератор конца списка
iterator erase(iterator i);	Удаляет элемент, на который указывает итератор i. Возвращает итератор элемента, который расположен следующим за удаленным
iterator erase(iterator *начало*, **iterator *конец);***	Удаляет элементы, заданные между итераторами *начало* и *конец.* Возвращает итератор элемента, который расположен следующим за последним удаленным
reference front(); **const_reference front() const;**	Возвращает ссылку на первый элемент списка
allocator_type get_allocator() const;	Возвращает распределитель памяти списка
iterator insert(iterator i, **const T &*значение* = T());**	Вставляет параметр *значение* перед элементом, заданным итератором *i.* Возвращает итератор элемента
void insert(iterator i, size_type *число*, **const T &*значение);***	Вставляет *число* копий параметра *значение* перед элементом, заданным итератором *i*
template<class inIter> void insert (iterator i, InIter *начало,* InIter *конец);*	Вставляет последовательность, определенную между итераторами *начало* и *конец,* перед элементом, заданным итератором i
size_type max_size() const;	Возвращает максимальное число элементов, которое может храниться в списке

Функция-член	Описание
void merge(list<T, Allocator> *&объект); **template<class Comp> void merge (list < T, Allocator> &*объект*, Comp *ф_сравн);**	Выполняет слияние упорядоченного списка, хранящегося в объекте *объект,* с вызывающим упорядоченным списком. Результат упорядочивается. После слияния список, хранящийся в объекте *объект* становится пустым. Во второй форме для определения того, является ли значение одного элемента меньшим, чем значение другого, может задаваться функция сравнения *ф_сравн*
void pop_back();	Удаляет последний элемент списка
void pop_front();	Удаляет первый элемент списка
void push_back(const T &*значение);	Добавляет в конец списка элемент, значение которого равно параметру *значение*
void push_front(const T &*значение);	Добавляет в начало списка элемент, значение которого равно параметру *значение*
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца списка
void remove(const T &*значение);	Удаляет из списка элементы, значения которых равны параметру *значение*
template<class UnPred> void remove_if(UnPred *пред);	Удаляет из списка значения, для которых истинно значение унарного предиката *пред*
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала списка
void resize(size_type *число*,T *значение* = T());	Изменяет размер списка в соответствии с параметром *число.* Если при этом список удлиняется, то добавляемые в конец списка элементы получают значение, заданное параметром *значение*
void reversed;	Выполняет реверс (т. е. реализует обратный порядок расположения элементов) вызывающего списка
size_type size() const;	Возвращает хранящееся на данный момент в списке число элементов
void sort(); **template<class Comp> void sort Comp *ф_сравн);**	Сортирует список. Во второй форме для определения того, является ли значение одного элемента меньшим, чем значение другого, может задаваться функция сравнения *ф_сравн*

Функция-член	Описание
void splice(iterator *i*, list<T, Allocator> *&объект);	Вставляет содержимое объекта *объект* в вызывающий список. Место вставки определяется итератором *i*. После выполнения операции *объект* становится пустым
void splice(iterator *i*, list<T, Allocator> *&объект*, iterator *элемент);	Удаляет элемент, на который указывает итератор *элемент,* из списка, хранящегося в объекте *объект,* и сохраняет его в вызывающем списке. Место вставки определяется итератором *i*
void splice(iterator *i*, list<T, Allocator> &*объект*, iterator *начало*, iterator *конец);	Удаляет диапазон элементов, обозначенный итераторами *начало* и *конец,* из списка, хранящегося в объекте *объект,* и сохраняет его в вызывающем списке. Место вставки определяется итератором *i*
void swap(list<T, Allocator> *&объект);	Обменивает элементы из вызывающего списка с элементами из объекта *объект*
void unique(); **template<class BinPred>** **void unique(BinPred *пред);**	Удаляет из вызывающего списка парные элементы. Во второй форме для выяснения уникальности элементов используется предикат *пред*

АССОЦИАТИВНЫЕ СПИСКИ

Шаблон для класса map:

```
template <class key, class T , class Comp=less<Key>,
          class Allocator = allocator <T>> class map
```

Ключевое слово *Allocator* задает распределитель памяти, который по умолчанию является стандартным. *Key* – данные типа ключ, *T* – тип данных, *Comp* – функция для сравнения двух ключей (по умолчанию стандартная объект-функция *less*()).

Определены следующие конструкторы:

```
explicit map(const Comp &ф_сравн = Comp(),
             const Allocator &a = Allocator());

map(const map<Key, T, Comp, Allocator>&объект);

template <class InIter>map(InIter начало, InIter конец,
                const Comp &ф_сравн = Comp(),
                const Allocator &a = Allocator());
```

Определены операторы сравнения:
`== < <= != > >=`

В ассоциативном списке хранятся пары ключ/значение в виде объектов типа *pair*.
Шаблон объекта *pair*:

```
template <class Ktype, class Vtype> struct pair
{
  typedef Ktype первый_тип;   // тип ключа
  typedef Vtype второй_тип;   // тип значения
          Ktype первый;       // содержит ключ
          Vtype второй;       // содержит значение
                              // конструкторы
          pair();
          pair(const Ktype &k, Vtype &v);
  template<class A, class B> pair(const  <A,B> &объект);
}
```

Создавать пары ключ/значение можно с помощью функции:

```
  template<class Ktype, class Vtype> pair(Ktype, Vtype>
           make_pair()(const Ktype &k, Vtype &v);
```

Функции-члены класса *map*

Функция-член	**Описание**
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента ассоциативного списка
void clear();	Удаляет все элементы ассоциативного списка
size.type count (const key_type *&k)* const;	Возвращает 1 или 0, в зависимости от того, встречается или нет в ассоциативном списке ключ *к*
bool empty() const;	Возвращает истину, если вызывающий ассоциативный список пуст, в противном случае возвращает ложь
iterator end(); **const.iterator end() const;**	Возвращает итератор конца ассоциативного списка
pair<iterator, iterator> **equal_range (const key_type pair** **<const_iterator, const_iterator>** **equal_range(const key_type *&k)* const;**	Возвращает пару итераторов, которые указывают на первый и последний элементы ассоциативного списка, содержащего указанный ключ *к*
void erase(iterator i);	Удаляет элемент, на который указывает итератор *i*
void erase (iterator *начало,*iterator *конец);	Удаляет элементы, заданные между итераторами *начало* и *конец*
size_type erase (const key_type *&k);	Удаляет элементы, соответствующие значению ключа *к*
iterator find. (const key_type *&k); **const_iterator find (const key_type *&k)* const;**	Возвращает итератор по заданному ключу *к.* Если ключ не обнаружен, возвращает итератор конца ассоциативного списка
allocator_type get_allocator() const;	Возвращает распределитель памяти ассоциативного списка
iterator insert(iterator *i,* **const value_type *&значение);**	Вставляет параметр *значение* на место элемента или после элемента, заданного итератором *i.* Возвращает итератор этого элемента
template<class lnIter> void insert **(InIter *начало,*InIter *конец);**	Вставляет последовательность элементов, заданную итераторами *начало* и *конец*
pair<iterator, bool>insert **(const value_type *&значение);**	Вставляет *значение* в вызывающий ассоциативный список. Возвращает итератор вставленного элемента. Элемент вставляется только в случае, если такого в ассоциативном списке еще нет. При удачной вставке элемента функция возвращает значение pair<iterator, true>, в противном случае — pair<iterator, false>

Функция-член	Описание
key_compare key_comp() const;	Возвращает объект-функцию сравнения ключей
iterator lower_bound (const key_type *&k); **const.iterator lower_bound** **(const key_type *&k)* const;**	Возвращает итератор первого элемента ассоциативного списка, ключ которого равен или больше заданного ключа *к*
size_type max_size() const;	Возвращает максимальное число элементов, которое можно хранить в ассоциативном списке
reference operator[] (const key_type &i);	Возвращает ссылку на элемент, соответствующий ключу *i* Если такого элемента не существует, он вставляется в ассоциативный список
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца ассоциативного списка
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала ассоциативного списка
size_type size() const;	Возвращает хранящееся на данный момент в ассоциативном списке число элементов
void swap(map<Key, T, Comp, Allocator> ***&объект);***	Обменивает элементы из вызывающего ассоциативного списка с элементами из объекта *объект*
iterator upper_bound **(const key_type *&k);** **const_iterator upper_bound** **(const key_type &k) const;**	Возвращает итератор первого элемента ассоциативного списка, ключ которого больше заданного ключа *к*
value_compare value_comp() const;	Возвращает объект-функцию сравнения значений

АЛГОРИТМЫ

Алгоритмы библиотеки стандартных шаблонов

Алгоритм	Назначение
adjacent_find	Выполняет поиск смежных парных элементов в последовательности. Возвращает итератор первой пары
binary_search	Выполняет бинарный поиск в упорядоченной последовательности
copy	Копирует последовательность
copy_backward	Аналогична функции copy(), за исключением того, что перемещает в начало последовательности элементы из ее конца
count	Возвращает число элементов в последовательности
count_if	Возвращает число элементов в последовательности, удовлетворяющих некоторому предикату
equal	Определяет идентичность двух диапазонов
equal_range	Возвращает диапазон, в который можно вставить элемент, не нарушив при этом порядок следования элементов в последовательности
fill	Заполняет диапазон заданным значением
find	Выполняет поиск диапазона для значения и возвращает первый найденный элемент
find_end	Выполняет поиск диапазона для подпоследовательности. Функция возвращает итератор конца подпоследовательности внутри диапазона
find_first_of	Находит первый элемент внутри последовательности, парный элементу внутри диапазона
find_if	Выполняет поиск диапазона для элемента, для которого определенный пользователем унарный предикат возвращает истину
for_each	Назначает функцию диапазону элементов
generate **generate_n**	Присваивает элементам в диапазоне значения, возвращаемые порождающей функцией
includes	Определяет, включает ли одна последовательность все элементы другой последовательности

Алгоритм	Назначение
inplace_merge	Выполняет слияние одного диапазона с другим. Оба диапазона должны быть отсортированы в порядке возрастания элементов. Результирующая последовательность сортируется
iter_swap	Меняет местами значения, на которые указывают два итератора, являющиеся аргументами функции
lexicographical_compare	Сравнивает две последовательности в алфавитном порядке
lower_bound	Обнаруживает первое значение в последовательности, которое не меньше заданного значения
make_heap	Выполняет пирамидальную сортировку последовательности (пирамида, на английском языке heap, — полное двоичное дерево, обладающее тем свойством, что значение каждого узла не меньше значения любого из его дочерних узлов.
max	Возвращает максимальное из двух значений
max_element	Возвращает итератор максимального элемента внутри диапазона
merge	Выполняет слияние двух упорядоченных последовательностей, а результат размещает в третьей последовательности
min	Возвращает минимальное из двух значений
min_element	Возвращает итератор минимального элемента внутри диапазона
mismatch	Обнаруживает первое несовпадение между элементами в двух последовательностях. Возвращает итераторы обоих несовпадающих элементов
next_permutation	Образует следующую перестановку (permutation) последовательности
nth_element	Упорядочивает последовательность таким образом, чтобы все элементы, меньшие заданного элемента Б, располагались перед ним, а все элементы, большие заданного элемента Е, — после него
partial_sort	Сортирует диапазон
partial_sort_copy	Сортирует диапазон, а затем копирует столько элементов, сколько войдет в результирующую последовательность
partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат

Алгоритм	Назначение
	возвращает ложь
pop_heap	Меняет местами первый и предыдущий перед последним элементы, а затем восстанавливает пираМИДу
prev_permutation	Образует предыдущую перестановку последовательности
push_heap	Размещает элемент на конце пирамиды
random_shuffle	Беспорядочно перемешивает последовательность
remove **remove_if** **remove_copy** **remove_copy_if**	Удаляет элементы из заданного диапазона
replace **replace jf** **replace_copy** **replace_copy_if**	Заменяет элементы внутри диапазона
reverse **reverse_copy**	Меняет порядок сортировки элементов диапазона на обратный
rotate **rotate_copy**	Выполняет циклический сдвиг влево элементов в диапазоне
search	Выполняет поиск подпоследовательности внутри последовательности
search_n	Выполняет поиск последовательности заданного числа одинаковых элементов
set_difference	Создает последовательность, которая содержит различающиеся участки двух упорядоченных наборов
set_intersection	Создает последовательность, которая содержит одинаковые участки двух упорядоченных наборов
set_symmetric_difference	Создает последовательность, которая содержит симметричные различающиеся участки двух упорядоченных наборов
set_union	Создает последовательность, которая содержит объединение (union) двух упорядоченных наборов
sort	Сортирует диапазон
sort_heap	Сортирует пирамиду внутри диапазона
stable_partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину,

Алгоритм	Назначение
	располагались перед элементами, для которых предикат возвращает ложь. Разбиение на разделы остается постоянным; относительный порядок расположения элементов последовательности не меняется
stable_sort	Сортирует диапазон. Одинаковые элементы не переставляются
swap	Меняет местами два значения
swap_ranges	Меняет местами элементы в диапазоне
transform	Назначает функцию диапазону элементов и сохраняет результат в новой последовательности
unique unique_copy	Удаляет повторяющиеся элементы из диапазона
upper_bound	Обнаруживает последнее значение в последовательности, которое не больше некоторого значения

ПРИМЕР 1. Библиотека стандартных шаблонов. Класс-контейнер vector

Основные операции вектора.

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

int main()
{
    vector<int> v; // создание вектора нулевой длины

    // вывод на экран размера исходного вектора v
    cout << "Размер = " << v.size() << endl;

    // помещение значений в конец вектора,
    // по мере необходимости вектор будет расти
    for (int i = 0; i < 10; i++)
       v.push_back(i);

    // вывод на экран текущего размера вектора v
    cout << "Новый размер = " << v.size() << endl;

    // вывод на экран содержимого вектора v
    // доступ к содержимому вектора
    // с использованием оператора индекса
    cout << "Текущее содержимое:\n";
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

    // помещение новых значений в конец вектора,
    // и опять по мере необходимости вектор будет расти
    for (int i = 0; i < 10; i++)
        v.push_back(i + 10);

    // вывод на экран текущего размера вектора
    cout << "Новый размер = " << v.size() << endl;

    // вывод на экран содержимого вектора
    cout << "Текущее содержимое:\n";
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
```

```
    // изменение содержимого вектора
    for (unsigned int i = 0; i < v.size(); i++)
        v[i] = v[i] + v[i];

    // вывод на экран содержимого вектора
    cout << "Удвоенное содержимое:\n";
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

    // доступ к вектору через итератор
    vector<int>::iterator p = v.begin();
    while (p != v.end))
    {
        cout << *p << " ";
        ++p; // префиксный инкремент итератора быстрее
             // постфиксного
    }

    return 0;
}
```

ПРИМЕР 2. Библиотека стандартных шаблонов. Класс-контейнер vector

Демонстрация функций inset() и erase().

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

int main()
{
    vector<int> v(5, 1); // создание пятиэлементного вектора
                         // из единиц

    // вывод на экран исходных размера и содержимого вектора
    cout << "Размер = " << v.size() << endl;
    cout << "Исходное содержимое:\n";
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

    vector<int>::iterator p = v.begin();
    p += 2;   // p указывает на третий элемент

    // вставка в вектор на то место,
    // куда указывает итератор p десяти новых элементов,
    // каждый из которых равен 9
    v.insert(p, 10, 9);

    // вывод на экран размера
    // и содержимого вектора после вставки
    cout << "Размер после вставки = " << v.size() << endl;
    cout << "Содержимое после вставки:\n";
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

    // удаление вставленных элементов
    p = v.begin();
    p += 2; // указывает на третий элемент


    v.erase(p, p + 10);
    // удаление следующих десяти элементов
    // за элементом, на который указывает итератор p
```

```
    // вывод на экран размера
    // и содержимого вектора после удаления
    cout << "Размер после удаления = " << v.size() << endl;
    cout << "Содержимое после удаления:\n";
    for (unsigned int i = 0; i < v.size(); i++)
       cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

ПРИМЕР 3. Библиотека стандартных шаблонов. Класс-контейнер vector

Хранение в векторе объектов пользовательского класса

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

class Demo
{
    double d;
public:
    Demo()
    {
        d = 0.0;
    }
    Demo(double x)
    {
        d = x;
    }
    Demo& operator=(double x)
    {
        d = x;
        return *this;
    }
    double getd() const
    {
        return d;
    }
};

bool operator<(const Demo& a, const Demo& b)
{
    return a.getd() < b.getd();
}

bool operator==(const Demo& a, const Demo& b)
{
    return a.getd() == b.getd();
}
```

```
int main()
{
    vector<Demo> v;

    for (int i = 0; i < 10; i++)
        v.push_back(Demo(i/3.0)); // добавить элемент
                                  // в конец вектора

    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i].getd() << " ";

    cout << endl;

    for (unsigned int i = 0; i < v.size(); i++)
        v[i] = v[i].getd() * 2.1;

    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i].getd() << " ";

    return 0;
}
```

ПРИМЕР 4. Библиотека стандартных шаблонов. Класс-контейнер list

Основные операции списка: создание, определение числа элементов, просмотр элементов с удалением.

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

int main()
{
    list<char> lst;    // создание пустого списка

    for (int i = 0; i < 10; i++)
        lst.push_back('A' + i); // добавить в конец списка

            // число элементов в списке
    cout << "Размер = " << lst.size() << endl;

    cout << "Содержимое: ";
    while (!lst.empty())  // пока список не пуст
    {
        list<char>::iterator p;
        p = lst.begin(); // итератор первого элемента списка
        cout << *p;
        lst.pop_front(); // удаление первого элемента списка
    }

    return 0;
}
```

ПРИМЕР 5. Библиотека стандартных шаблонов. Класс-контейнер list

Основные операции списка: создание, определение числа элементов, просмотр элементов без удаления.

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

int main()
{
    list<char> lst;

    for (int i = 0; i < 10; i++)
       lst.push_back('A' + i); // добавить в конец списка

    // число элементов в списке
    cout << "Размер = " << lst.size() << endl;

    // итератор на первый элемент в списке
    list<char>::iterator p = lst.begin();

    cout << "Содержимое: ";
    while (p != lst.end()) // итератор на элемент, следующий
    {                      // за последним в списке
        cout << *p;
        ++p;
    }

    return 0;
}
```

ПРИМЕР 6. Библиотека стандартных шаблонов. Класс-контейнер list

Элементы можно размещать не только начиная с начала списка, но также и начиная с его конца. Создается два списка, причем во втором порядок организации элементов обратный первому.

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

int main()
{
    list<char> lst;

    for (int i = 0; i < 10; i++)
        lst.push_back('A' + i); // добавить в конец списка

    cout << "Размер прямого списка = " << lst.size()
                                       << endl;
    cout << "Содержимое прямого списка: ";

    // Удаление элементов из первого списка
    // и размещение их в обратном порядке во втором списке
    list<char> revlst;
    while (!lst.empty())
    {
        char element = lst.front(); // получить первый
                                    // элемент в списке
        cout << element;
        lst.pop_front(); // удаление первого элемента списка
        revlst.push_front(element); // добавить в начало
                                    // списка
    }
    cout << endl;

    cout << "Размер обратного списка = ";
    cout << revlst.size() << endl;
    cout << "Содержимое обратного списка: ";
    for (list<char>::iterator p = revlst.begin();
                              p != revlst.end(); ++p)
        cout << *p;

    return 0;
}
```

ПРИМЕР 7. Библиотека стандартных шаблонов. Класс-контейнер list

Сортировка списка.

```
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
    list<char> lst;

    // заполнение списка случайными символами
    for (int i = 0; i < 10; i++)
        lst.push_back('A' + (rand()%26));

    cout << "Исходное содержимое: ";
    list<char>::iterator p = lst.begin();
    while (p != lst.end())
    {
        cout << *p;
        ++p;
    }
    cout << endl;

    // сортировка списка
    lst.sort();

    cout << "Отсортированное содержимое: ";
    p = lst.begin();
    while (p != lst.end())
    {
        cout << *p;
        ++p;
    }

    return 0;
}
```

ПРИМЕР 8. Библиотека стандартных шаблонов. Класс-контейнер list

Слияние двух списков

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

int main()
{
    list<char> lst1, lst2;
    int n = 0, m = 0;

    cin>> n >> m;
    for (int i = 0; i < n; i += 2)
        lst1.push_back('A' + i);
    for (int i = 1; i < m; i += 2)
        lst2.push_back('A' + i);

    cout << "Содержимое первого списка: ";
    list<char>::iterator p = lst1.begin();
    while (p != lst1.end())
    {
        cout << *p;
        ++p;
    }
    cout << endl;

    cout << "Содержимое второго списка: ";
    p = lst2.begin();
    while (p != lst2.end())
    {
        cout << *p;
        ++p;
    }
    cout << endl;
```

```
    // Слияние двух списков
    lst1.merge(lst2);

    if (lst2.empty())
        cout << "Теперь второй список пуст\n";

    cout << "Содержимое первого списка после слияния:\n";
    p = lst1.begin();
    while (p != lst1.end())
    {
        cout << *p;
        ++p;
    }

    return 0;
}
```

ПРИМЕР 9. Библиотека стандартных шаблонов. Класс-контейнер list

Использование в списке объектов пользовательского класса

```
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

const int MAX = 40;
class Project
{
    char name[MAX];
    int days_to_completion;
public:

    Project()
    {
        strcpy_s(name, MAX, " ");
        days_to_completion = 0;
    }
    Project(const char* n, int d)
    {
        strcpy_s(name, MAX, n);
        days_to_completion = d;
    }
    void add_days(int i)
    {
        days_to_completion += i;
    }
    void sub_days(int i)
    {
        days_to_completion -= i;
    }
    bool completed() const
    {
        return !days_to_completion;
    }
    void report() const
    {
        cout << name << ": ";
        cout << days_to_completion;
        cout << " day to finish" << endl;
    }
};
```

```
int main()
{
    list<Project> proj;
    proj.push_back(Project("compile", 35));
    proj.push_back(Project("exel", 190));
    proj.push_back(Project("STL", 1000));

    // вывод проектов на экран
    for (list<Project>::iterator p = proj.begin();
                                 p != proj.end(); ++p)
        p->report();

    // увеличение сроков выполнения первого проекта
    // на 10 дней
    list<Project>::iterator p = proj.begin();
    p->add_days(10);

    // последовательное завершение первого проекта
    do
    {
        p->sub_days(5);
        p->report();
    } while (!p->completed());

    return 0;
}
```

ПРИМЕР 10. Библиотека стандартных шаблонов. Класс-контейнер map

Иллюстрация возможностей ассоциативного списка.

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

int main()
{
    map<char, int> m;

    // размещение пар в ассоциативном списке
    for (int i = 0; i < 10; i++)
        m.insert(pair<char, int>('A' + i, i));

    char ch = 0;
    cout << "Введите ключ: ";
    cin >> ch;

    map<char, int>::iterator p;
    // поиск значения по заданному ключу
    p = m.find(ch);
    if (p != m.end())
        cout << p -> second;
    else
        cout << "Такого ключа в ассоциативном списке нет\n";

    return 0;
}
```

ПРИМЕР 11. Библиотека стандартных шаблонов. Класс-контейнер map

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

int main()
{
    map<char, int> m;

    // размещение пар в ассоциативном списке
    for (int i = 0; i < 10; i++)
    {
        m.insert(make_pair(char)('A' + i, i));
    }

    char ch;
    cout << "Введите ключ: ";
    cin >> ch;

    map<char, int>::iterator p;
            // поиск значения по заданному ключу
    p = m.find(ch);
    if (p != m.end())
        cout << p->second;
    else
        cout << "Такого ключа в ассоциативном списке нет\n";

    return 0;
}
```

ПРИМЕР 12. Библиотека стандартных шаблонов. Класс-контейнер map

Ассоциативный список слов и антонимов

```
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class word
{
    char str[20];
public:
    word() { strcpy(str, ""); }
    word(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

// для объектов типа word определим оператор < (меньше),
// чтобы его можно было использовать как ключ
// в контейнере map
bool operator<(word a, word b)
{
    return strcmp(a.get(), b.get()) < 0;
}



class opposite
{
    char str[20];
public:
    opposite() { strcmp(str, ""); }
    opposite(char *s) { strcpy(str, s); }
    char *get() { return str; }
};
```

```
int main()
{
    map<word, opposite> m;

    // размещение в ассоциативном списке слов и антонимов
    m.insert(pair<word, opposite>(word("да"),
                                  opposite("нет")));
    m.insert(pair<word, opposite>(word("хорошо"),
                                 opposite("плохо")));
    m.insert(pair<word, opposite>(word("влево"),
                                  opposite("вправо")));
    m.insert(pair<word, opposite>(word("вверх"),
                                  opposite("вниз")));

    // поиск антонима по заданному слову
    char str[80];
    cout << "Введите слово: ";
    cin >> str;

    map<word, opposite>::iterator p;

    p = m.find(word(str));
    if (p != m.end())
        cout << "Антоним: " << p->second.get();
    else
        cout << "Такого слова в ассоциативном списке нет\n";
    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    return 0;
}
```

ПРИМЕР 13. Библиотека стандартных шаблонов. Алгоритмы count() и count_if()

Демонстрация алгоритмов count и count_if.

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

/* это унарный предикат, который определяет,
   является ли значение четным  */
bool even(int x)
{
    return !(x%2);
}

int main()
{
    vector<int> v;

    for (int i = 0; i < 20; i++)
    {
        if (i%2)
            v.push_back(1);
        else
            v.push_back(2);
    }

    cout << "Последовательность: ";
    for (int i = 0; i < v.size(); i++)
       cout << v[i] << " ";
    cout << endl;

    int n = count(v.begin(), v.end(), 1);
    cout << n << " элементов равно 1\n";

    n = count_if(v.begin(), v.end(), even);
    cout << n << " четных элементов\n";

    return 0;
}
```

ПРИМЕР 14. Библиотека стандартных шаблонов. Алгоритм remove_copy

Демонстрация алгоритма remove_copy

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

int main()
{
    vector<int> v, v2(20);

    for (int i = 0; i < 20; i++)
    {
        if (i%2) v.push_back(1);
        else v.push_back(2);
    }

    cout << "Последовательность: ";
    for (int i = 0; i < v.size(); i++)
         cout << v[i] << " ";
    cout << endl;

             // удаление единиц
    remove_copy(v.begin(), v.end(), v2.begin(), 1);

    cout << "Результат: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << " ";
    cout << endl;

    return 0;
}
```

ПРИМЕР 15. Библиотека стандартных шаблонов. Алгоритм reverse

Демонстрация алгоритма reverse

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

int main()
{
    vector<int> v;

    for (int i = 0; i < 10; i++)
        v.push_back(i);

    cout << "Исходная последовательность: ";
    for (int i = 0; i < v.size(); i++)
       cout << v[i] << " ";
    cout << endl;

    reverse(v.begin(), v.end());

    cout << "Обратная последовательность: ";
    for (int i = 0; i < v.size(); i++)
       cout << v[i] << " ";

    return 0;
}
```

ПРИМЕР 16. Библиотека стандартных шаблонов. Алгоритм transform

Пример использования алгоритма transform

```
#include <iostream>
#include <list>
#include <algoritm>
using namespace std;
                          // Простая функция модификации
int xform(int i)
{
    return i * i;         // квадрат исходного значения
}

int main()
{
    list<int> x1;

    // размещение значений в списке
    for (int i = 0; i < 10; i++)
       v.push_back(i);

    cout << "Исходное содержимое списка x1: ";
    list<int>::iterator p = x1.begin();
    while (p != x1.end())
    {
        cout << *p << " ";
        ++p;
    }

    cout << endl;

    // модификация элементов списка x1
    p = transform(x1.begin(), x1.end(), x1.begin(), xform);

    cout << "Модифицированное содержимое списка x1: ";
    p = x1.begin();
    while (p != x1.end())
    {
        cout << *p << " ";
        ++p;
    }
    return 0;
}
```