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 дисциплине

«ПРОГРАММИРОВАНИЕ НА C++»

2024
МИНСК

Практическое руководство к лабораторному практикуму «БИБЛИОТЕКА STL» по дисциплине «ПРОГРАММИРОВАНИЕ НА C++» предназначено для студентов, изучающих базовый курс программирования на языке C++, специальностей «Компьютерная безопасность», «Прикладная информатика», «Радиофизика».

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

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

СОДЕРЖАНИЕ

Библиотека стандартных шаблонов STL **5**
Строки стандартного класса string **7**
Методы для работы со строками класса string 7
Операции для работы со строками класса string 10
Векторы **11**
Списки **13**
Ассоциативные списки **16**
Алгоритмы **18**
Алгоритм вычисления выражений через обратную польскую запись **21**
Строки string **23**
ПРИМЕР 1. Объявление и инициализация строки string 23
ПРИМЕР 2. Инициализация строки string. Оператор =. Метод assign 23
ПРИМЕР 3. Ввод строки string. Оператор >> 24
ПРИМЕР 4. Ввод строки string. Метод getline 24
ПРИМЕР 5. Длина строки string. Методы length, size 24
ПРИМЕР 6. Доступ к элементу строки string. Оператор []. Метод at 25
ПРИМЕР 7. Сравнение строк. Операторы сравнения 25
ПРИМЕР 8. Сравнение строк. Метод compare 26
ПРИМЕР 9. Объединение строк. Оператор +. Метод append 27
ПРИМЕР 10. Вставка строки (подстроки) в строку. Метод insert 27
ПРИМЕР 11. Замена строки (подстроки) в строке. Метод replace 28
ПРИМЕР 12. Удаление подстроки в строке. Метод erase 28
ПРИМЕР 13. Выделение подстроки в строке. Метод substr 28
ПРИМЕР 14. Обмен содержимого строк. Метод swap, reverse 29
ПРИМЕР 15. Поиск подстроки в строке. Метод find 29
ПРИМЕР 16. Поиск символа подстроки в строке. Метод find_first_of 30
ПРИМЕР 17. Поиск символа подстроки в строке. Метод find_first_not_of 30
ПРИМЕР 18. Выделение лексем. Методы find_first_not_of, find_first_of 31
ПРИМЕР 19. Выделение лексем. Чтение из потока 32
ПРИМЕР 20. Строки С и С++. Методы copy, c_str 32
Класс-контейнер vector **33**
ПРИМЕР 21. Класс-контейнер vector. Основные операции 33
ПРИМЕР 22. Класс-контейнер vector. Методы insert и erase 34
ПРИМЕР 23. Получение элементов вектора через итераторы 34
ПРИМЕР 24. Получение элементов вектора через итераторы. Range-based for. 35
ПРИМЕР 25. Вставка, удаление элементов вектора 36
ПРИМЕР 26. Класс-контейнер vector. Хранение объектов пользовательского класса 37
ПРИМЕР 27. Класс-контейнер vector. Матрица (вектор векторов) 38
Класс-контейнер list **40**
ПРИМЕР 28. Класс-контейнер list. Создание, определение числа элементов, просмотр с удалением 40
ПРИМЕР 29. Класс-контейнер list. Просмотр элементов списка в прямом и обратном порядке 40
ПРИМЕР 30. Класс-контейнер list. Вставка и удаление элементов списка 41
ПРИМЕР 31. Класс-контейнер list. Добавление элементов в конец и начало списка 41
ПРИМЕР 32. Класс-контейнер list. Сортировка элементов списка 42
ПРИМЕР 33. Класс-контейнер list. Слияние списков 43
ПРИМЕР 34. Класс-контейнер list. Использование пользовательского класса 43
Класс-контейнер map **45**
ПРИМЕР 35. Класс-контейнер map. Создание, поиск 45
ПРИМЕР 36. Класс-контейнер map. Алгоритм find 45
Класс-контейнер set **46**

ПРИМЕР 37. Класс-контейнер set (упорядоченное множество). Вставка, удаление, поиск 46
Алгоритмы 47
ПРИМЕР 38. Алгоритмы count и count_if 47
ПРИМЕР 39. Алгоритм remove_copy 48
ПРИМЕР 40. Алгоритм reverse.....48
ПРИМЕР 41. Алгоритм transform.....49

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 двусвязный 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	Стек на основе динамического массива	<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

СТРОКИ СТАНДАРТНОГО КЛАССА STRING

Язык С++ включает в себя новый класс, называемый ***string*.** Этот класс во многом улучшает традиционный строковый тип, позволяет обрабатывать строки также как данные других типов, а именно с помощью операторов. Он более эффективен и безопасен в использовании, не нужно заботиться о создании массива нужного размера для строковой переменной, класс *string* берет на себя ответственность за управлением памятью. Если при создании приложения скорость выполнения не является доминирующим фактором, класс *string* предоставляет безопасный и удобный способ обработки строк. Для работы со строками класса *string* существует множество **методов и операций.**

Объявление и инициализация строк string.

```
                    //создаем пустую строку
     string s1;     //вызов конструктора без аргументов
                   // создаем строку из С-строки
     string s2("aaaa");//вызов конструктора с одним аргументом
//   string s2 = "aaaa";   // или так
            // создаем строку из С-строки 10 символов
            //вызов конструктора с двумя аргументами
     string s3("abcdefghijklmnopqrstuvwxyz",10);
     string s4(5,'!'); //создаем строку из 5 одинаковых символов
     string  s5(s3); // создаем строку-копию из строки s3
                       // создаем строку-копию из строки s3
     string  s6(s3,5,3); //  начиная с индекса 5 не более 3 символов
```

Методы для работы со строками класса string

	МЕТОД	ЗАПИСЬ	ОПИСАНИЕ
1	**at**	`at (unsigned  n)`	доступ к n-му элементу строки
2	**append**	`append (string &str)`	добавляет строку str к концу вызывающей строки (тоже, что оператор +)
		`append (string &str, unsigned  pos, unsigned  n);`	добавляет к вызывающей строке n символов строки str, начиная с позиции pos
		`append (char  *sr, unsigned  n);`	добавляет к вызывающей строке n символов С-строки s
3	**assign**	`assign (string &str)`	присваивает строку str вызывающей строке (тоже, что s2=s1)
		`assign ( string &str, unsigned  pos, unsigned  n);`	присваивает вызывающей строке n символов строки str, начиная с позиции pos
		`assign (char *sr, unsigned  n);`	присваивает вызывающей строке n символов С-строки s
4	**capacity**	`unsigned int capacity ();`	возвращает объем памяти, занимаемый строкой
5	**compare**	`int compare (string &str);`	сравнение двух строк, возвращает значение <0, если вызывающая строка

			лексикографически меньше str, =0, если строки равны и >0, если вызывающая строка больше
		`int compare (string &str, unsigned pos, unsigned n);`	сравнение со строкой str n символов вызывающей строки, начиная с позиции pos; возвращает значение <0, если вызывающая строка лексикографически меньше str, =0, если строки равны и >0, если вызывающая строка больше
		`int compare (unsigned pos1, unsigned n1, string &str, unsigned pos2, unsigned n2);`	n1 символов вызывающей строки, начиная с позиции pos1, сравниваются с подстрокой строки str длиной n2 символов, начиная с позиции pos2; возвращает значение <0, если вызывающая строка лексикографически меньше str, =0, если строки равны и >0, если вызывающая строка больше
6	**copy**	`unsigned int copy (char *s, unsigned n, unsigned pos = 0);`	копирует в символьный массив s n элементов вызывающей строки, начиная с позиции pos; нуль-символ в результирующий массив не заносится; метод возвращает количество скопированных элементов
7	**c_str**	`char * c_str()`	возвращает указатель на С-строку, содержащую копию вызываемой строки; полученную С-строку нельзя изменить
8	**empty**	`bool empty();`	возвращает истину, если строка пустая
9	**erase**	`erase (unsigned pos = 0, unsigned n = npos);`	удаляет n элементов, начиная с позиции pos (если n не задано, то удаляется весь остаток строки) npos-самое большое число >0 типа unsigned
10	**find**	`unsigned int find (string &str, unsigned pos = 0);`	ищет самое левое вхождение строки str в вызывающей строке, начиная с позиции pos; возвращает позицию вхождения, или npos(самое большое число >0 типа unsigned , если вхождение не найдено
		`unsigned find (char c, unsigned pos = 0);`	ищет самое левое вхождение символа с в вызывающей строке, начиная с позиции pos; возвращает позицию вхождения, или npos(самое большое число >0 типа unsigned, если вхождение не найдено
		`unsigned rfind (char c, unsigned pos = 0);`	ищет самое правое вхождение символа с в вызывающей строке, начиная с позиции pos; возвращает позицию вхождения, или npos(самое большое число >0 типа unsigned, если вхождение не найдено
11	**find_first_of**	`unsigned find_first_of (string &str, unsigned pos = 0);`	ищет самое левое вхождение любого символа строки str в вызывающей

			строке, начиная с позиции pos; возвращает позицию вхождения, или npos(самое большое число >0 типа unsigned, если вхождение не найдено
12	**find_last_ of**	`unsigned find_last_of (string &str, unsigned pos = 0);`	ищет самое правое вхождение любого символа строки str в вызывающей строке, начиная с позиции pos; возвращает позицию вхождения, или npos(самое большое число >0 типа unsigned, если вхождение не найдено
13	**find_first_ not_of**	`unsigned find_first_not_of (string &str, unsigned pos = 0);`	ищет самый левый символ не равный ни одному символу из строки str в вызывающей строке, начиная с позиции pos; возвращает позицию вхождения, или npos(самое большое число >0 типа unsigned, если вхождение не найдено
14	**insert**	`insert (unsigned pos, string &str);`	вставляет строку str в вызывающую строку, начиная с позиции pos
		`insert (unsigned pos1, string &str, unsigned pos2, unsigned n);`	вставляет в вызывающую строку, начиная с позиции pos1 n символов строки str, начиная с позиции pos2
		`insert (unsigned pos, char * sr, unsigned n);`	вставляет в вызывающую строку n символов С-строки s**,** начиная с позиции pos
15	**length**	`unsigned length ();`	возвращает размер строки
16	**max_size**	`unsigned max_size();`	возвращает максимальную длину строки
17	**replace**	`replace (unsigned pos, unsigned n, string &str);`	заменяет n элементов, начиная с позиции pos вызывающей строки, элементами строки str
		`replace (unsigned pos1, unsigned n1, string &str, unsigned pos2, unsigned n2);`	заменяет n1 элементов, начиная с позиции pos1 вызывающей строки, n2 элементами строки str, начиная с позиции pos2
		`replace (unsigned pos, unsigned n1, char *s, unsigned n2);`	заменяет n1 элементов, начиная с позиции pos вызывающей строки, n2 элементами С-строки s
18	**size**	`unsigned size();`	возвращает размер строки
19	**substr**	`string substr (unsigned pos = 0, unsigned n = npos);`	выделяет подстроку длиной n из исходной строки, начиная с позиции pos
20	**swap**	`swap (string &str)`	обменивает содержимое вызывающей строки и строки str

Операции для работы со строками класса string

	оператор	**ЗАПИСЬ** **string s1,s2,s3;**	**ОПИСАНИЕ**
1	+	`s3 = s1+ s2;`	конкатенация (сцепление) строк, можно присоединить единичный символ к строке
2	+=	`s1 += s2;`	конкатенация (сцепление) строк с присвоением результата
3	=	`s2 = s1;`	присваивание
4	==	`s2 == s1`	лексикографическое сравнение на равенство строк
5	!=	`s2 != s1`	лексикографическое сравнение на неравенство строк
6	>	`s2 > s1`	лексикографическое сравнение строк на >
7	>=	`s2 >= s1`	лексикографическое сравнение строк на >=
8	<	`s2 < s1`	лексикографическое сравнение строк на <
9	<=	`s2 <= s1`	лексикографическое сравнение строк на <=
10	[]	`s1[i]`	индексация (обращение к элементу строки)
11	>>	`cin >> s1;`	ввод строки (лучше метод getline)
12	<<	`cout << s2;`	вывод строки

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:
`== < <= != > >=`

Функции-члены класса *vector*

`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
`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*

`template<class InIter>` `void assign(InIter 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

`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*

`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*

`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 inlter>` `void insert (inlter начало,inlter 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>
`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

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
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 , располагались перед ним, а все элементы, большие заданного элемента, — 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 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, 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

АЛГОРИТМ ВЫЧИСЛЕНИЯ ВЫРАЖЕНИЙ ЧЕРЕЗ ОБРАТНУЮ ПОЛЬСКУЮ ЗАПИСЬ

Обычные алгебраические (логические, математические) выражения можно записывать в виде **обратной польской нотации** – записи без скобок. Нотация – «польская» в честь польского математика Яна Лукашевича, который её предложил. Нотация – «обратная» из-за того, что порядок операндов и операций обратный относительно исходного.

Пример:

Выражение		Обратная польская нотация
A + (B - C) * D – F / (G + H) =	→	A B C – D * + F G H + / - =

Если A = 6 B = 4 C = 1 D = 2 F = 3 G = 7 H = 5.
Тогда обратная польская нотация: 6 4 1 – 2 * + 3 7 5 + / - =

Алгоритм вычисления выражения.

Вычислить обратное польское выражение проще, чем обычное алгебраическое.

Просматриваем все элементы слева направо. Как только встречается операция, выполняем ее по отношению к двум предыдущим элементам, заменяя два элемента одним.

6 4 1 – 2 * + 3 7 5 + / - = выполняем 4-1=3 →
6 3 2 * + 3 7 5 + / - = выполняем 3*2=6 →
6 6 + 3 7 5 + / - = выполняем 6+6=12 →
12 3 7 5 + / - = выполняем 7+5=12 →
12 3 12 / - = выполняем 3/12=0.25 →
12 0.25 - = выполняем 12-0.25=11.75 →
11.75 = результат

Алгоритм преобразования выражения в обратную польскую нотацию.

Необходимо использовать два стека (списка) X и Y.

Необходимо определить приоритеты операций. Например, так:

операции	приоритет
* /	3
+ -	2
(	1
=	0

1. Просматриваем выражение слева направо. Операнды помещаем в стек X , левые скобки и операции в стек Y.
2. Встретив правую скобку, отыскиваем в стеке соответствующую ей левую. При этом все, что сверху – выталкивается из стека Y и заносится в стек X.
3. Если приоритет очередной операции меньше приоритета операции вершины стека , то операции из вершины стека Y выталкиваются в стек X, пока не найдём операцию с приоритетом ниже, либо пока стек не окажется пустым .
4. Результат оказывается с стеке X.

Пример преобразования выражения в обратную польскую запись

A + (B - C) * D – F / (G + H) =

	стек **X**	стек **Y**
1. A	A	
2.+	A	+
3. (	A	(→ +
4.B	B → A	(→ +
5. -	B → A	- → (→ +
6. C)	C → B → A	- → (→ +
7.)	- → C → B → A	+
8. *	- → C → B → A	* → +
9. D	D → - → C → B → A	* → +
10. –	+ → * → D → - → C → B → A	-
11. F	F → + → * → D → - → C → B → A	-
12. /	F → + → * → D → - → C → B → A	/ → -
13. (	F → + → * → D → - → C → B → A	(→ /→ -
14. G	G → F → + → * → D → - → C → B → A	(→ / → -
15. +	G → F → + → * → D → - → C → B → A	+ → (→ / → -
16. H	H → G → F → + → * → D → - → C → B → A	+→ (→ /→ -
17.)	+ → H → G → F → +→ *→ D → - → C → B → A	/ → -
18. =	- → / → + → H → G → F → + → *→ D→ - → C→ B→ A	

результат в обратном порядке

A B C – D * + F G H + / - =

СТРОКИ STRING

ПРИМЕР 1. Объявление и инициализация строки string

```
#include <iostream>
#include <string>                  // строковый класс
using namespace std;
void main()
{
     string s1;              //создаем пустую строку
     string s2("aaaa");  // создаем строку из С-строки
               // создаем строку из С-строки 10 символов
     string s3("abcdefghijklmnopqrstuvwxyz",10);
               // создаем строку из 5 одинаковых символов
     string s4(5,'!');
               // создаем строку-копию из строки s3
     string  s5(s3);
               // создаем строку-копию из строки s3
              //  начиная с индекса 5 не более 3 символов
     string  s6(s3,5,3);

     cout<<"s1= "<<s1<<endl;
     cout<<"s2= "<<s2<<endl;
     cout<<"s3= "<<s3<<endl;
     cout<<"s4= "<<s4<<endl;
     cout<<"s5= "<<s5<<endl;
     cout<<"s6= "<<s6<<endl;
}
```

ПРИМЕР 2. Инициализация строки string. Оператор =. Метод assign

```
#include <iostream>
#include <string>                  // строковый класс
using namespace std;
void main()
{    string s1, s2, s3;
// в классе string определены три оператора присваивания:
// string & operator = (const string& str);
// string & operator = (const char* s);
// string & operator = (char c);

     s1 = '1';
     s2 = "bbbbbb";
     s3 = s2;
     cout<<"s1= "<<s1<<endl;
     cout<<"s2= "<<s2<<endl;
     cout<<"s3= "<<s3<<endl;
                              //   метод assign
     s2.assign("ccccccc");     //  s2="ccccccc"
     s3.assign(s2);            //  s3=s2
     cout<<"s1 = "<<s1<<endl;
     cout<<"s2 = "<<s2<<endl;
     cout<<"s3 = "<<s3<<endl;
```

```
      s2.assign("1234");          //  s2=”1234”
            // в s3 из s2 3 символа, начиная с 1 позиции
      s3.assign(s2,1,3);
      cout<<"s2= "<<s2<<endl;
      cout<<"s3= "<<s3<<endl;

      char s[]="56789";
                  //  присваивает  s3  3 символа С-строки
      s3.assign(s,3);
      cout<<"s= "<<s<<endl;
      cout<<"s3= "<<s3<<endl;
}
```

ПРИМЕР 3. Ввод строки string. Оператор >>

```
#include <iostream>
#include <string>
using namespace std;
void main()
{     string s1;
      cout << "   Enter a string: ";
      //ввод выполняется до первого пробельного символа
      cin >> s1;
      cout << "You entered: " << s1 << endl;
}
```

ПРИМЕР 4. Ввод строки string. Метод getline

```
#include <iostream>
#include <string>
using namespace std;
void main()
{     string s1;
      cout << "Enter a string: ";
      getline(cin,s1);                  //ввод строки
      cout << "You entered: " << s1 << endl;
      cin.get();       // удаление из потока символа ‘\n’.
      cout << "   Enter a string: ";
                // свой разделитель для ввода строки
      getline(cin,s1,'&');
      cout << "You entered: " << s1 << endl;
      cin.get();       // удаление из потока символа ‘&’.
}
```

ПРИМЕР 5. Длина строки string. Методы length, size

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

```
void main()
{    string st("********************");
     cout << " " << st.length()<< " " << st.size() <<
            "  "<<st.max_size()<<endl;
     st = "Good Morning";
     cout << " " << st.length()<< " " << st.size() <<
            "  "<<st.max_size()<<endl;
     st = "Hello";
     cout << " " << st.length()<< " " << st.size() <<
            "  "<<st.max_size()<<endl;
}
```

ПРИМЕР 6. Доступ к элементу строки string. Оператор []. Метод at

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string st("********************");
     for (int i = 0; i < st.length(); i++)
          cout << st[i];
                    // если i выходит за пределы строки,
                    // то поведение не определено
     st = "Good Morning";
     for (int i = 0; i < st.length(); i++)
          cout << st.at(i);
          // если i выходит за пределы строки,
          // метод возвращает исключение типа out_of_rang
     st = "Hello";
     for (int i = 0; i < st.length(); i++)
          cout << st[i];
}
```

ПРИМЕР 7. Сравнение строк. Операторы сравнения

Вводим строки в цикле, выход – ввод “пустой” сроки.
Вывод на экран результатов сравнения двух строк.

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string st1, st2;
     cout << "Enter a string \n";
     getline(cin,st1);          //  ввод 1 строки

     while (true)
     {    cout << "Enter a string \n";
          getline(cin,st2);      //  ввод 2 строки
          cin.get();
          if ( st2 == "")
            break;               //  выход из цикла
          cout  << endl<< st2 ;
```

```
            // операторы лексикографического сравнения строк
        if (st2 == st1)
         cout << "  =  ";
        else
             if (st2 < st1)
             cout <<"  <  ";
             else
             cout  << " > ";
        cout  << st1<<endl ;
        st1 = st2;
    }
}
```

ПРИМЕР 8. Сравнение строк. Метод compare

Вводим строки в цикле, выход – ввод “пустой” сроки.
Вывод на экран результатов сравнения двух строк.

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string st1, st2;
     cout << "Enter a string \n";
     getline(cin,st1);          //  ввод 1 строки
     while (true)
     {    cout << "Enter a string \n";
          cin.get();
          getline(cin,st2);          //  ввод 2 строки
          if ( !st2.compare("")) break; // выход из цикла
          cout  << endl<< st2 ;

               // лексикографическое сравнение строк
          if (!st2.compare(st1)) cout <<"  =  ";
          else
               if (st2.compare(st1)<0) cout <<"  <  ";
               else      cout  << " > ";
          cout  << st1<<endl ;
          cout  << endl <<st2[1]<<st2[2];

                    // лексикографическое сравнение подстрок
          if (!st2.compare(1,2,st1,2,3))
           cout <<"  =  ";
          else
               if (st2.compare(1,2,st1,2,3)<0)
               cout <<"  <  ";
               else
                cout  << " > ";
          cout  << st1[2]<<st1[3]<<st1[4]<<endl ;
          st1 = st2;
     }
}
```

ПРИМЕР 9. Объединение строк. Оператор +. Метод append

```
#include <iostream>
#include <string>
using namespace std;
void main()
{     string s1("11");
      string s2("2222");
      string s3 ("333333");
      string s4("44444444");
      s1 += s2;                        // добавить s2 к s1
      cout<<"s1 = s1 + s2 = "<<s1<<endl;
      s4 = s1 + s2;                   // добавить s1 к s2
      cout<<"s4 = s1 + s2 = "<<s4<<endl;
      s4 = s4 + '!';                 // добавить s1 к s2
      cout<<"s4 = s4 + ! = "<<s4<<endl;
      s2.append(s1);                // добавить s1 к s2
      cout<<"s2 + s1 = "<< s2<<endl;
      //  добавить к  s3  2 символа строки s1 со 1 позиции
      s3.append(s4,1,2);
      cout<<"s3 + s4 = "<< s3<<endl;
      char s[] = "56789";
                  //  добавить к  s3  2 символа С-строки s
      s3.append(s,2);
      cout<<"s3 + s = "<< s3<<endl;
}
```

ПРИМЕР 10. Вставка строки (подстроки) в строку. Метод insert

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
      string s1("1111111");
      string s2("23456789");
      s1.insert(3,s2);     // вставить s2 в s1 с 3 позиции
      cout<<"s1="<< s1<<endl;
      s1 = "1111111";
      s2 = "23456789";
   // вставить 4 символа s2 со 2 позиции в s1 с 3 позиции
      s1.insert(3,s2,2,4);
      cout<<"s1="<< s1<<endl;
      s1 = "1111111";
      char s[] = "23456789";
            // вставить 4 символа s  в s1 с 3 позиции
      s1.insert(3,s,4);
      cout<<"s1 = "<< s1<<endl;
}
```

ПРИМЕР 11. Замена строки (подстроки) в строке. Метод replace

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("1111111");
     string s2("23456789");
  // замена  2 символов в s1 с 3 позиции элементами s2
     s1.replace(3,2,s2);
     cout<<"s1="<< s1<<endl;
     s1 = "1111111";
     s2 = "23456789";
     // замена  2 символов в s1 с 3 позиции 1 символом
    //  из 4 позиции строки  s2
     s1.replace(3,2,s2,4,1);
     cout<<"s1="<< s1<<endl;
     s1 = "1111111";
     char s[] = "23456789";
    // замена  2 символов в s1 с 3 позиции 4 символами s
     s1.replace(3,2,s,4);
     cout<<"s1="<< s1<<endl;
}
```

ПРИМЕР 12. Удаление подстроки в строке. Метод erase

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("123456789");
          // удаление   2 символов в s1 с 3 позиции
     s1.erase(3,2);
     cout<<"s1="<< s1<<endl;
          // удаление   всех символов в s1 с 3 позиции
    s1 = "123456789";
     s1.erase(3);
     cout<<"s1="<< s1<<endl;
     s1 = "123456789";
     s1.erase();          // удаление   всех символов  s1
     cout<<"s1="<< s1<<endl;
}
```

ПРИМЕР 13. Выделение подстроки в строке. Метод substr

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("123456789");
     string s2;
         // s2 - подстрока  s1 из 2 символов  с 3 позиции
     s2 = s1.substr(3,2);
     cout<<"s2="<< s2<<endl;
     s1 = "123456789";
```

```
          // s2 – подстрока s1 всех символов  с 3 позиции
     s2 = s1.substr(3);
     cout<<"s2="<< s2<<endl;
     s1 = "123456789";
     s2 = s1.substr();                        // s2=s1
     cout<<"s2="<< s2<<endl;
}
```

ПРИМЕР 14. Обмен содержимого строк. Метод swap, reverse

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("123456789");
     string s2("abcdef");
     s1.swap(s2);
     cout<<"s1="<< s1<<endl;
     cout<<"s2="<< s2<<endl;
    string s3("12345678");
     reverse(s3.begin(),s3.end());
     cout<< s3 <<endl;
}
```

ПРИМЕР 15. Поиск подстроки в строке. Метод find

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("123123123");
     string s2("12");
     unsigned k;
      //  поиск подстроки s2 в строке s1 с 4 позиции
     k = s1.find(s2,4);
     cout<<"   "<< k<<endl;
      //  поиск подстроки s2 в строке s1 с 7 позиции
    k = s1.find(s2,7);
     cout<<"   "<< k<<endl;
        //  поиск символа '1'  в строке s1 с 4 позиции
    k = s1.find('1',4);
     cout<<"   "<< k<<endl;
       //  поиск символа '1'  в строке s1 с 4 позиции
     k = s1.rfind('1',4);
     cout<<"   "<< k<<endl;
}
```

ПРИМЕР 16. Поиск символа подстроки в строке. Метод find_first_of

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("1234561728");
     string s2("12");
     char s3[100]="124";
     unsigned k;
     // поиск первого любого символа из подстроки s2 в строке s1
     // с 4 позиции
     k = s1.find_first_of(s2, 4);
     cout << "   " << k << endl;                    // 6
     //  поиск первого любого символа из count первых
     // символов подстроки s3 в строке s1  с 4 позиции
     k = s1.find_first_of(s3, 4, 2);
     cout << "   "<< k<<endl;                  // 6
     // поиск первого любого символа из подстроки s3 в
     // строке s1 с 4 позиции
     k = s1.find_first_of(s3, 4);
     cout<<"   "<< k<<endl;          // 6
     //  поиск символа '1'  в строке s1 с 4 позиции
     k = s1.find_first_of('1', 4);
     cout << "   "<< k << endl;                    // 6
}
```

ПРИМЕР 17. Поиск символа подстроки в строке. Метод find_first_not_of

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string s1("1234561728");
     string s2("12");
     char s3[100]="124";
     unsigned k;
     // поиск первого символа отличного от любого символа
     // из подстроки s2 в строке s1 с 6 позиции
     k = s1.find_first_not_of(s2, 6);
     cout<<"   "<< k<<endl;             // 7
     // поиск первого символа отличного от любого символа
     // из count первых символов из подстроки s3
     // в строке s1 с 6 позиции
     k = s1. find_first_not_of (s3, 6, 2);
     cout<<"   "<< k<<endl;             // 7
     // поиск первого любого символа отличного от любого символа
     // из подстроки s3 в строке s1 с 6 позиции
     k = s1. find_first_not_of (s3, 6);
     cout<<"   "<< k<<endl;          // 7
     // поиск первого символа отличного от символа '1'
     // в строке s1 с 6 позиции
     k = s1.find_first_not_of ('1', 6);
     cout << "   " << k << endl;                    // 7
}
```

ПРИМЕР 18. Выделение лексем. Методы find_first_not_of, find_first_of

```
#include <iostream>
#include <string>
using namespace std;
#define OUT

void lexem(string str, string delim, OUT string& res);

void main()
{
  string delim(" .,;!?-:");        // строка разделителей
  string s1, s2;
  cout << "Enter a string \n";
  getline(cin,s1);                 //  ввод  строки
  lexem(s1, delim, OUT s2);
  cout <<  s2 << endl;            //  вывод результатов
}

void lexem(string str, string delim, OUT string& res)
{
  unsigned int wordBegin = 0, wordEnd = 0;

                      // позиция начала лексемы
  wordBegin = str.find_first_not_of(delim, wordEnd);
                      // позиция конца лексемы
  wordEnd = str.find_first_of(delim, wordBegin);

  if (wordEnd >= str.length())
    wordEnd = str.length();

  while (wordBegin < str.length())
  {               // выделение лексемы
    string word = str.substr(wordBegin, wordEnd - wordBegin);
    if (res.length())
        res += " ";
    res += word;                          // результирующая строка
                      // позиция начала лексемы
    wordBegin = str.find_first_not_of(delim, wordEnd);
                      // позиция конца лексемы
    wordEnd = str.find_first_of(delim, wordBegin);

    if (wordEnd >= str.length())
      wordEnd = str.length();
  }
}
```

ПРИМЕР 19. Выделение лексем. Чтение из потока

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
#define OUT

void lexem(string str, string delim, OUT string& res)
{    // заменяем все символы в строке str из delim на пробелы
     for (int i = 0; i < str.length(); i++)
     {
          char c = str[i];
          if (delim.find(str[i]) < delim.length())
               str[i] = ' ';
     }
     stringstream stream(str);
     string word;
     while (stream >> word)  // читаем из потока следующую лексему
     {
          if (res.length())
               res += ' ';
          res += word;
     }
}
void main()
{
     string delim(" .,;!?-:");        // строка разделителей
     string s1, s2;
     cout << "Enter a string \n";
     getline(cin,s1);                 //  ввод  строки
     lexem(s1, delim, OUT s2);
     cout <<  s2 << endl;            //  вывод результатов
}
```

ПРИМЕР 20. Строки С и С++. Методы copy, c_str

```
#include <iostream>
#include <string>
using namespace std;
void main()
{    string str("1234567890");
     char s[80];
     int k;
     // копируем 3 символа str с 5 позиции в s
     k = str.copy(s, 3, 5);
     cout<<s<<"   "<<k<<endl;
     // копируем 3 символа str с 5 позиции в s
     k = str.copy(s, 3, 5);
     s[3] = '\0';
     cout << s << "   " << k << endl;
     cout << str.c_str() << endl;
}
```

КЛАСС-КОНТЕЙНЕР VECTOR

ПРИМЕР 21. Класс-контейнер vector. Основные операции

Основные операции над вектором.

```
#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 22. Класс-контейнер vector. Методы insert и erase

Демонстрация функций вставки и удаления элементов.

```
#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 размераи 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 23. Получение элементов вектора через итераторы

```
#include <vector>
#include <iostream>
using namespace std;
int main(void)
{    vector<int> vect;
    vect.push_back(2);
    vect.push_back(3);
    vect.push_back(5);
```

```
    vect.push_back(6);
    for (vector<int>::iterator p = vect.begin();
        p != vect.end(); ++p)
        cout << *p;
    cout << endl;
/* Ключевое слово auto вместо типа переменной говорит компилятору
определить тип переменной исходя из присваиваемого ей при
инициализации значения.  */
    for (auto p = vect.begin(); p != vect.end(); ++p)
        cout << *p;
    cout << endl;
/* Можно воспользоваться итератором идущим в обратную сторону. Для
этого получаем итератор начала и конца диапазона методами rbegin и
rend, инкремент итератора сдвигает его в обратную сторону*/
    for (vector<int>::reverse_iterator p = vect.rbegin();
        p != vect.rend(); ++p)
        cout << *p;
    cout << endl;
/* Константный итератор позволяет читать элементы контейнера, но не
позволяет их изменять. У константного вектора можно получить только
константный итератор. */
    const vector<int>& c_vect = vect;
    for (vector<int>::const_iterator p = c_vect.begin();
        p != c_vect.end(); ++p)
        cout << *p;
    cout << endl;
/* также можно просмотреть элементы константного контейнера и в
обратную сторону */
    for (vector<int>::const_reverse_iterator
        p = c_vect.rbegin(); p != c_vect.rend(); ++p)
        cout << *p;
    cout << endl;
}
```

ПРИМЕР 24. Получение элементов вектора через итераторы. Range-based for.

Range-based for позволяет компактнее записывать циклы, перебирающие элементы контейнеров! Дословный перевод для "Range-based for" - "for по диапазону".

```
#include <vector>
#include <iostream>
using namespace std;
int main(void)
{    vector<float> vect;
    vect.push_back(2.0);
    vect.push_back(3.0);
    vect.push_back(5.0);
    vect.push_back(6.0);

/* Три цикла делают одно и то же и компилируются в один и тот же
машинный код.*/
    for (vector<float>::iterator p = begin(vect);
                                  p != end(vect); ++p)
    {   float val = *p;   cout << val << " ";    }
```

```
    cout << endl;

    for (float val : vect)  cout << val << " ";
    cout << endl;

    for (auto val : vect) cout << val << " ";
    cout << endl;

    vector<string> vectS;
    vectS.push_back("2.0");
    vectS.push_back("3.0");
    vectS.push_back("5.0");
    vectS.push_back("6.0");
    for (auto& val : vectS)  cout << val << " ";
    cout << endl;
    for (const auto& val : vectS) cout << val << " ";
    cout << endl;
}
```

ПРИМЕР 25. Вставка, удаление элементов вектора

```
#include <vector>
#include <iostream>
using namespace std;
int main(void)
{   vector<float> vect;
    vect.push_back(2.0);    vect.push_back(3.0);
    vect.push_back(5.0);    vect.push_back(6.0);
              // Удалить последний элемент в стеке
    vect.pop_back();
    for (auto val : vect)
        cout << val;
    cout << endl;
/* Вставка элемента 4 в произвольную позицию.  Первый параметр - итератор
на позицию куда произвести вставку (вставка осуществляется перед этим
элементом, на который указывает итератор). Некоторые итераторы позволяют
перемещаться не только на следующий элемент, но и сразу на несколько
vect.begin() - итератор на первый элемент вектора, vect.begin()+1 -
итератор смещённый на второй элемент вектора (второй, если считать с 1),
vect.begin()+2 - итератор смещённый на третий элемент вектора. Именно в
эту позицию (перед третьим элементом) и будет произведена вставка.*/
    vect.insert(vect.begin() + 2, 4);
    for (auto val : vect)   cout << val;
    cout << endl;
/*  Удаление одного или нескольких элементов вектора в произвольной
позиции. vect.begin()+1 - итератор смещённый на второй элемент вектора
(счёт с 1). vect.end() - итератор указывающий на элемент следующий за
последним.  vect.end()-1 - итератор указывающий на последний элемент.
Диапазон задаваемый итераторами включает первый элемент и не включает
последний, поэтому будут удалены элементы на позициях(счёт с 1): со
второй до предпоследней, то есть второй и третий элементы.*/
    vect.erase(vect.begin() + 1, vect.end() - 1);
    for (auto val : vect)  cout << val;
    cout << endl;
```

```
/* Изменить размер вектора. (удалить лишние элементы в конце, или
дополнить до указанного размера, второй параметр – элемент
заполнитель)*/
    vect.resize(5, 1);
    for (auto val : vect)
        cout << val;
    cout << endl;
/* Удаление всех элементов из вектора. Память остаётся выделенной,
поэтому добавление новых элементов после этого не потребует её
перевыделения.*/
    vect.clear();
    for (auto val : vect)
        cout << val;
    cout << endl;
}
```

ПРИМЕР 26. Класс-контейнер vector. Хранение объектов пользовательского класса

Хранение в векторе объектов пользовательского класса.

```
#include <iostream>
#include <vector>
using namespace std;
class Demo      // пользовательский класс
{    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;    }
};
               // операция <
bool operator<(const Demo& a, const Demo& b)
{    return a.getd() < b.getd();   }
                 // операция =
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 27. Класс-контейнер vector. Матрица (вектор векторов)

Создание матрицы как вектора векторов.

```
#include <iostream>
#include <vector>
using namespace std;
// функция вывода матрицы на экран
void vyvod(vector<vector<int>> vec)
{    for (int i = 0; i < vec.size(); i++)// Цикл по строкам
    {                    // Цикл по элементам в строке
        for (int j = 0; j < vec[i].size(); j++)
            cout << vec[i][j] << ' ';
              // Вывод элементов i строки вектора
        cout << endl;
    }
    cout << endl;
}
        // функция инициализации матрицы случайными числами
void vvod(vector<vector<int>> &vec, int m)
{    for (int i = 0; i < vec.size(); i++)// Цикл по строкам
    {                    // Цикл по элементам в строке
        for (int j = 0; j < vec[i].size(); j++)
            vec[i][j] = rand() % m;
    }
}
        // функция  инициализации вектора случайными числами
void vvod_vec(vector<int> &vec, int m)
{   for (int j = 0; j < vec.size(); j++)
       vec[j] = rand() % m;
}
// ------------------------------------------------
int main()
{    int n, m; // переменные, отвечающие за размер матрицы
    // -------------------------- 1 -------------------
    cin >> n >> m;              // Ввод размеров матрицы
    vector<vector<int>> vec1;  // нет памяти для матрицы
    for (int i = 0; i < n; i++)
    {
        vector <int> temp;      // вектор-строка (памяти нет)
        for (int j = 0; j < m; j++)     // добавляем элементы
            temp.push_back(rand() % m);// в строку
        vec1.push_back(temp);  // добавляем строку в матрицу
    }
    vyvod(vec1);               // вывод матрицы
   // --------------------------  2 --------------------
    cin >> n >> m;              // Ввод размеров матрицы
    vector<vector<int>> vec2;  // нет памяти для матрицы
    for (int i = 0; i < n; i++)
    {   vector<int> temp(m);   // одномерный вектор-строка
```

```
                                // из m элементов
    vvod_vec(temp, m);          // инициализация
    vec2.push_back(temp);       // добавляем строку в матрицу
}
vyvod(vec2);                  // вывод матрицы
// ---------------------------   3 -------------------
cin >> n >> m;  // Ввод размеров матрицы
vector<vector<int>> vec3(n, vector <int>(m, 0));
    // Объявление матрицы на n строк по m элементов
    // заполнение 0
vyvod(vec3);  // вывод матрицы
vvod(vec3, m); // инициализация
vyvod(vec3);  // вывод матрицы
// ---------------------------    4 ---------------
cin >> n >> m;  // Ввод размеров матрицы
vector<vector<int>> vec4(n, vector <int>(m));
// Объявление матрицы на n строк по m элементов
vvod(vec4,m);  // инициализация
vyvod(vec4);  // вывод матрицы
// ---------------------------   5    ---------------
cin >> n >> m;  // Ввод размеров векторов
vector<vector<int>> vec5(n);
    // Объявление матрицы на n строк
for (size_t i = 0; i < n; ++i)
{    // для каждой строки выделяем память на m элементов
    vec5[i] = vector<int>(m);
    vvod_vec(vec5[i],m);      // инициализация
}
vyvod(vec5);   // вывод матрицы
// ---------------------------   6    ---------------
cin >> n >> m;  // Ввод размеров векторов
vector < vector <int> > vec6(n);
    // Объявление матрицы на n строк
for (size_t i = 0; i < n; ++i)
{    // для каждой строки выделяем память на m элементов
    vec6[i].resize(m);
    vvod_vec(vec6[i],m);   // инициализация
}
vyvod(vec6);   // вывод матрицы
// ---------------------------   7  ---------------
// вывод элементов массива через итераторы
cin >> n >> m;  // Ввод размеров векторов
vector<vector<int>> vec7(n, vector<int>(m, 1));
    // Объявление матрицы на n строк по m элементов
                  // заполнение 1
for (vector<vector<int>>::iterator it1 = vec7.begin();
                              it1 != vec7.end(); ++it1)
{
    for (vector<int>::iterator it2 = (*it1).begin();
                              it2 != (*it1).end(); ++it2)
        cout << *(it2) << ' ';//Вывод элементов i строки
    cout << endl;
}
return 0; }
```

КЛАСС-КОНТЕЙНЕР LIST

ПРИМЕР 28. Класс-контейнер list. Создание, определение числа элементов, просмотр с удалением

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;
                          // удаление списка
    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 29. Класс-контейнер list. Просмотр элементов списка в прямом и обратном порядке

Просмотр элементов списка в прямом и обратном порядке 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 << "Size = " << lst.size() << endl;
                           // просмотр элементов списка
    cout << "Содержимое: ";
                          // итератор первого элемента списка
    list<char>::iterator p = lst.begin();
    cout << "---->: ";
    while(p != lst.end())    // итератор конца списка
    {   cout << *p;
        ++p;
    }
    cout << endl;
    // просмотр элементов списка в обратном порядке
    // итератор последнего элемента списка
```

```
    list<char>::reverse_iterator rp = lst.rbegin();
    cout << "<----: ";
    while(rp != lst.rend())     // итератор конца списка
    {   cout << *rp;
        ++rp;  // это строка не меняется,
               // обратный итератор задаёт направление
    }
    cout << endl;
    return 0;
}
```

ПРИМЕР 30. Класс-контейнер list. Вставка и удаление элементов списка

```
#include <list>
#include <iostream>
using namespace std;
int main(void)
{  list<int> lst;
/* lst.front() и lst.back() возвращают ссылки, то есть первый и
последний элемент листа можно не только читать, но и изменять.
Применять осторожно, если лист пустой то при обращении к этим методам
будет сгенерировано исключение throw*/
    lst.push_back(3);
    lst.push_back(4);
    lst.push_front(2);
    lst.push_front(1);

    for (list<int>::iterator p = lst.begin();
                             p != lst.end(); ++p)
        cout << *p;
    cout << endl;

    lst.pop_back();
    lst.pop_front();
    cout << lst.front();
    cout << lst.back();
    lst.clear();
}
```

ПРИМЕР 31. Класс-контейнер list. Добавление элементов в конец и начало списка

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())
    {       // получить первый элемент в списке
      char element = lst.front();
      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 32. Класс-контейнер list. Сортировка элементов списка

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;

    lst.sort();// сортировка списка

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

ПРИМЕР 33. Класс-контейнер list. Слияние списков

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;

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

    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 34. Класс-контейнер list. Использование пользовательского класса

Использование в списке объектов пользовательского класса.

```
#include <iostream>
#include <list>
#include <cstring>
using namespace std;
const int MAX = 40;
```

```
class Project          // пользовательский класс
{     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
      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;
}
```

КЛАСС-КОНТЕЙНЕР MAP

ПРИМЕР 35. Класс-контейнер map. Создание, поиск

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 36. Класс-контейнер map. Алгоритм find

Ассоциативный список слов и антонимов.

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

class word      // пользовательский класс слов 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   // пользовательский класс слов opposite
{    char str[40];
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;
}
```

КЛАСС-КОНТЕЙНЕР SET

ПРИМЕР 37. Класс-контейнер set (упорядоченное множество). Вставка, удаление, поиск

```
#include <set>
#include <list>
#include <iostream>
using namespace std;
int main(void)
{
    list<int> lst;
    lst.push_back(3);
    lst.push_back(4);
    lst.push_front(2);
    lst.push_front(1);
    for (list<int>::iterator p = lst.begin();
        p != lst.end(); ++p)
        cout << *p;
    cout << endl;

    set<int> tree;
                    // Вставить один элемент в множество
    tree.insert(6);
/* Вставить в множество все элементы из другого контейнера (в
двухсвязном списке lst находятся элементы 1 2 3 4).*/
    tree.insert(lst.begin(), lst.end());
```

```
/* Вывод элементов с помощью итератора – так же, как и в любом другом
контейнере, но элементы в контейнере set сразу хранятся в
упорядоченном виде*/
    for (set<int>::iterator p = tree.begin();
        p != tree.end(); ++p)
        cout << *p;
    cout << endl;
    //  тот же вывод элементов,но через ranged-for
    for (auto val : tree)
        cout << val;
    cout << endl;
/* Для проверки есть ли элемент в множестве  удобно использовать функцию
подсчитывающую количество элементов с заданным значением*/
    if (tree.count(3))  cout << "Has 3" << endl;
    if (tree.count(5)) cout << "Has 5" << endl;
       //Удалить из множества узел с указанным значением
    tree.erase(3);
    for (auto val : tree)
        cout << val;
    cout << endl;
/* возвращает итератор на элемент с указанным значением или tree.end(),
если такого значения нет в дереве.  удалить элемент, используя итератор*/
    set<int>::iterator p4 = tree.find(4);
    if (p4 != tree.end())
    {   cout << *p4;
        tree.erase(p4);
    }
    for (auto val : tree)
        cout << val;
    cout << endl;
}
```

АЛГОРИТМЫ

ПРИМЕР 38. Алгоритмы count и count_if

Демонстрация алгоритмов count и count_if.

```
#include <iostream>
#include <vector>
#include <algoritm>
using namespace std;
// 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 << " количество элементов равных 1\n";
    n = count_if(v.begin(), v.end(), even);
    cout << n << " количество четных элементов\n";
    return 0;
}
```

ПРИМЕР 39. 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 40. 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 41. 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;
}
```