

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Белгородский государственный технологический университет  
им. В.Г. Шухова

Утверждено  
научно-методическим  
советом университета

# **ИНФОРМАТИКА**

Методические указания к выполнению  
лабораторных работ и РГЗ для студентов очной формы обучения  
направлений бакалавриата  
09.03.02 - Информационные системы и технологии и  
09.03.03 - Прикладная информатика

Белгород  
2024

## **СОДЕРЖАНИЕ**

<b>Лабораторная работа №1 СИСТЕМЫ СЧИСЛЕНИЯ .....</b>	<b>6</b>
<b>Лабораторная работа №2 АЛГЕБРА ЛОГИКИ.....</b>	<b>10</b>
<b>Лабораторная работа №3 РАБОТА В СРЕДЕ MICROSOFT VISUAL STUDIO. РЕАЛИЗАЦИЯ ЛИНЕЙНЫХ И ВЕТВЯЩИХСЯ АЛГОРИТМОВ СРЕДСТВАМИ ЯЗЫКА C++ .....</b>	<b>14</b>
<b>Лабораторная работа №4 РЕАЛИЗАЦИЯ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ СРЕДСТВАМИ ЯЗЫКА C++ .....</b>	<b>39</b>
<b>Лабораторная работа №5 ОБРАБОТКА ОДНОМЕРНЫХ МАССИВОВ.....</b>	<b>47</b>
<b>Лабораторная работа №6 ОБРАБОТКА ДВУМЕРНЫХ МАССИВОВ. ФАЙЛОВЫЙ ВВОД- ВЫВОД. ....</b>	<b>55</b>

**Лабораторная работа №7 ПРИМЕНЕНИЕ  
ИТЕРАТИВНЫХ И РЕКУРСИВНЫХ ФУНКЦИЙ.....82**

**Лабораторная работа №8 ПОБИТОВЫЕ  
ОПЕРАЦИИ ЯЗЫКА C++ ..... 103**

**Лабораторная работа №9 СТАНДАРТНАЯ  
БИБЛИОТЕКА ШАБЛОНОВ STL. КЛАСС STRING.  
ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРНЫЕ  
КЛАССЫ ..... 112**

## **ПРАВИЛА ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ**

По курсу информатики предусмотрено выполнение ряда лабораторных работ. Студент обязан перед выполнением каждой лабораторной работы самостоятельно ознакомиться с теоретическим материалом и по ее результатам предоставить отчет. Все отчеты о выполнении лабораторных работ оформляются в отдельной тетради. Допускается оформлять отчеты в печатном виде на листах формата А4 или во время дистанционной формы проведения занятий сдавать на проверку файлы с необходимыми материалами. Отчет к лабораторным работам должен содержать:

1. Заголовок лабораторной работы – номер работы, данные о студенте, слова «Выполнение» и «Защита», название и цель работы.
2. Содержание работы (общее для всех студентов) и индивидуальное задание, которое выбирается из вариантов.
3. Блок-схемы разработанных алгоритмов (рекомендуется использовать Microsoft Visio).
4. Тексты программ на языке C++.
5. Результаты тестирования программ.
6. Вывод о выполненной работе.

## **Пример оформления лабораторной работы**

*Лабораторная работа №1*

*студента группы ИТ-222*

*Петрова Ильи Александровича*

*Выполнение: \_\_\_\_\_ Защита: \_\_\_\_\_*

### *Системы счисления*

*Цель работы: приобрести навыки перевода чисел из одной системы счисления в другую и выполнения арифметических действий в различных системах счисления.*

### *Содержание работы*

- 1. Ознакомиться с материалом, посвященным работе в двоичной, восьмеричной и шестнадцатеричной системах счисления.*
- 2. Создать новую книгу MS Excel. На первом листе разработать таблицу для перевода восьмизначного двоичного числа в восьмеричную и шестнадцатеричную системы счисления.*
- 3. ...*

### *Ход работы*

- 1. Прочел страницы ... учебника под редакцией ... Ознакомился с материалом методического пособия ...*
- 2. ...*

*Вывод: ...*

# Лабораторная работа №1

## СИСТЕМЫ СЧИСЛЕНИЯ

**Цель работы:** приобрести навыки перевода чисел из одной системы счисления в другую и выполнения арифметических действий в различных системах счисления.

### Содержание работы

1. Ознакомиться с материалом, посвященным работе в двоичной, восьмеричной и шестнадцатеричной системах счисления.
2. Создать новую книгу MS Excel. На первом листе разработать таблицу для перевода восьмизначного двоичного числа в восьмеричную и шестнадцатеричную системы счисления. Пример приведен на рисунке 1.1.

	A	B	C	D	E	F	G	H	I
1									
2	разряды	7	6	5	4	3	2	1	0
3	двоичное число	1	0	0	0	0	1	0	1
4	восьмеричное число	2	0	5					
5	шестнадцатеричное число	8	5						
6									
7									

Рис. 1.1 Пример таблицы для преобразования двоичного числа

Для каждого разряда двоичного числа необходимо использовать отдельную ячейку. После ввода двоичного числа значения коэффициентов восьмеричного и шестнадцатеричного чисел должны вычисляться средствами MS Excel.

3. На втором листе разработать таблицу для суммирования двух восьмизначных чисел в системе счисления с указанным основанием N ( $1 < N < 10$ ). Пример приведен на рисунке 1.2.

	A	B	C	D	E	F	G	H	I	J
1		основание системы счисления	3							
2										
3		перенос	0	0	0	0	1	1	1	0
4		+	1	1	0	1	0	1	2	2
5			0	0	0	0	0	1	1	1
6			1	1	0	1	1	0	1	0

Рис. 1.2 Таблица для сложения чисел

Основание системы счисления задавать в отдельной ячейке (на рисунке 1.2 ячейка C1). В диапазоне C3:J3 указать перенос в старшие разряды. Предусмотреть проверку коэффициентов вводимых чисел. Пример сообщения о недопустимом коэффициенте 4 для троичной системы счисления приведен на рисунке 1.3 в ячейке H2.

	A	B	C	D	E	F	G	H	I	J
1		основание системы счисления	3							
2								Ошибка		
3		перенос	0	0	0	0	1	1	1	0
4		+	1	1	0	1	0	4	2	2
5			0	0	0	0	0	1	1	1
6			1	1	0	1	1	3	1	0

Рис. 1.3 Сообщение об ошибке

4. На третьем листе разработать таблицу для нахождения разности двух восьмиразрядных чисел в системе счисления с указанным основанием N ( $1 < N < 10$ ). Пример приведен на рисунке 1.4.

Предусмотреть, кроме проверки корректности вводимых чисел, индикацию займа у старших разрядов.

	A	B	C	D	E	F	G	H	I	J	
1	основание системы счисления		3								
2											
3		займ	0	3	3	3	3	3	3	3	
4		вычит	-1	-1	-1	-1	-1	-1	-1	0	
5		-	1	0	0	0	0	0	0	0	
6			0	0	0	0	0	0	1	1	
7			0	2	2	2	2	2	1	2	
8											

Рис. 1.4 Таблица для нахождения разности чисел.

5. Разработать таблицу для перевода десятичного числа, не превосходящего 1024, в систему счисления с указанным основанием N (1 < N < 10). Разрешено использовать дополнительные ячейки. В случае неверных входных данных, выдать сообщение об ошибке. Пример такой таблицы приведен на рисунке 1.5.

	A	B	C	D	E	F	G	H	I	J	K	L	
1													
2	число	1024	основание	2									
3													
6		1	0	0	0	0	0	0	0	0	0	0	
7													

Рис. 1.5 Пример перевода десятичного числа с двоичную систему счисления

6. Оформить отчет по проделанной работе с примерами использованных выражений и полученными результатами. В конце сделать вывод.

### Контрольные вопросы

1. Что называется системой счисления?



2. Какие системы счисления называются непозиционными? Приведите пример такой системы счисления и записи чисел в ней?
3. Какие системы счисления называются позиционными?
4. Что называется основанием системы счисления?
5. Какие числа можно использовать в качестве основания системы счисления?
6. Какие системы счисления применяются в компьютере для представления информации?
7. Охарактеризуйте двоичную систему счисления: алфавит, основание системы счисления, запись числа, правила выполнения арифметических операций.
8. Почему двоичная система счисления используется в информатике?
9. Охарактеризуйте восьмеричную систему счисления: алфавит, основание системы счисления, запись числа, правила выполнения арифметических операций.
10. Дайте характеристику шестнадцатеричной системе счисления: алфавит, основание, запись чисел. Приведите примеры арифметических операций.
11. Сформулируйте правила перевода чисел из системы счисления с основанием  $n$  в десятичную систему счисления и обратного перевода: из десятичной системы счисления в систему счисления с основанием  $q$ .
12. Как выполнить перевод чисел из двоичной системы счисления в восьмеричную и обратно?
13. Как выполнить перевод из двоичной системы счисления в шестнадцатеричную и обратно?
14. По каким правилам выполняется перевод из восьмеричной в шестнадцатеричную систему счисления и наоборот?

## Лабораторная работа №2

### АЛГЕБРА ЛОГИКИ

**Цель работы:** научиться представлять логические функции, заданные таблицами истинности, в виде СДНФ и СКНФ.

#### Содержание работы

1. Ознакомиться с материалом, посвященным логическим функциям и законам алгебры логики.
2. Построить полную таблицу истинности для заданной функции алгебры логики. Смотри пример на рисунке 2.1.

	A	B	C	D	
1	$F = \neg x_3 \wedge x_1 \vee x_1 \wedge x_2$				
2					
3	$x_1$	$x_2$	$x_3$	F	
4	0	0	0	0	
5	0	0	1	0	
6	0	1	0	0	
7	0	1	1	0	
8	1	0	0	1	
9	1	0	1	0	
10	1	1	0	1	
11	1	1	1	1	

Рис. 2.1 Таблица истинности.

Вариант задания	Номер студента в журнале	Логическая функция
1.	1, 16	$(x_1 \rightarrow x_2) \wedge \neg x_3$
2.	2, 17	$(x_1 \rightarrow x_2) \wedge x_3$

Вариант задания	Номер студента в журнале	Логическая функция
3.	3, 18	$(\neg x_1 \rightarrow x_2) \wedge x_3$
4.	4, 19	$(x_1 \rightarrow \neg x_2) \wedge x_3$
5.	5, 20	$(\neg x_1 \rightarrow \neg x_2) \wedge x_3$
6.	6, 21	$(\neg x_1 \wedge \neg x_2) \rightarrow x_3$
7.	7, 22	$(\neg x_1 \rightarrow x_2) \wedge \neg x_3$
8.	8, 23	$(\neg x_1 \vee \neg x_2) \rightarrow \neg x_3$
9.	9, 24	$(x_1 \wedge x_2) \rightarrow \neg x_3$
10.	10, 25	$(x_1 \vee x_2) \rightarrow \neg x_3$
11.	11, 26	$(x_1 \vee \neg x_2) \rightarrow x_3$
12.	12, 27	$(\neg x_1 \vee x_2) \rightarrow x_3$
13.	13, 28	$(\neg x_1 \vee x_2) \rightarrow \neg x_3$
14.	14, 29	$(\neg x_1 \vee \neg x_2) \rightarrow \neg x_3$
15.	15, 30	$(x_1 \rightarrow \neg x_2) \vee x_3$

3. Представить заданную функцию в виде СДНФ. Для этого найти минтермы для отдельных строк таблицы истинности, а затем объединить их в ответе. Пример приведен на рисунке 2.2.

$x_1$	$x_2$	$x_3$	F	СДНФ
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	1	$x_1 \cdot \neg x_2 \cdot \neg x_3$
1	0	1	0	
1	1	0	1	$x_1 \cdot x_2 \cdot \neg x_3$
1	1	1	1	$x_1 \cdot x_2 \cdot x_3$
СДНФ F= $x_1 \cdot \neg x_2 \cdot \neg x_3 + x_1 \cdot x_2 \cdot \neg x_3 + x_1 \cdot x_2 \cdot x_3$				

Рис. 2.2 Функция в виде СДНФ.

4. Представить заданную функцию в виде СКНФ. Для этого найти макстермы для отдельных строк таблицы истинности, а затем объединить их в ответе. Пример приведен на рисунке 2.3.

$x_1$	$x_2$	$x_3$	F	СКНФ
0	0	0	0	$(x_1 + x_2 + x_3)$
0	0	1	0	$(x_1 + x_2 + \neg x_3)$
0	1	0	0	$(x_1 + \neg x_2 + x_3)$
0	1	1	0	$(x_1 + \neg x_2 + \neg x_3)$
1	0	0	1	
1	0	1	0	$(\neg x_1 + x_2 + \neg x_3)$
1	1	0	1	
1	1	1	1	
СКНФ F=	$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \neg x_3) \cdot (x_1 + \neg x_2 + x_3) \cdot (x_1 + \neg x_2 + \neg x_3) \cdot (\neg x_1 + x_2 + \neg x_3)$			

Рис. 2.3 Функция в виде СКНФ.

5. Оформить отчет по проделанной работе с примерами использованных выражений и полученными результатами. В конце сделать вывод.

## Контрольные вопросы

1. Что изучает наука логика?

2. Перечислите основные логические операции над высказываниями.
3. Какими символами обозначаются логические операции: отрицание, дизъюнкция, конъюнкция, импликация, эквивалентность?
4. Приведите таблицу истинности логических операции: конъюнкция, дизъюнкция, импликация, эквивалентность.
5. Каков порядок логических операций при преобразовании логических выражений?
6. Как определяется количество строк и столбцов в таблице истинности логического выражения?
7. Перечислите известные Вам логические функции для одной переменной.
8. Перечислите известные Вам логические функции для двух переменных.
9. Перечислите основные (базовые) логические функции.
10. Запишите известные Вам свойства элементарных функций алгебры логики в случае одной переменной.
11. Запишите известные Вам законы для элементарных функций алгебры логики в случае нескольких переменных.
12. В чем отличие ДНФ от СДНФ?
13. В чем отличие КНФ от СКНФ?
14. Каков алгоритм перехода от табличного вида логической функции к СДНФ?
15. Каков алгоритм перехода от табличного вида логической функции к СКНФ?
16. Как связаны СДНФ и СКНФ одной функции алгебры логики?

# **Лабораторная работа №3**

## **РАБОТА В СРЕДЕ MICROSOFT VISUAL STUDIO.**

### **РЕАЛИЗАЦИЯ ЛИНЕЙНЫХ И ВЕТВЯЩИХСЯ АЛГОРИТМОВ СРЕДСТВАМИ ЯЗЫКА C++**

**Цель работы:** получить навыки в создании, настройке и отладке консольных приложений на языке программирования C++ в среде Visual Studio; ознакомиться с базовыми средствами ввода-вывода.

### **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

#### **Среда разработки программ Microsoft Visual Studio**

Среда Visual Studio позволяет работать с несколькими языками программирования, в т.ч. и с C++. Создаваемые приложения могут разрабатываться как в виде неуправляемого кода для платформы Win32, так и управляемого для платформы .Net. Далее рассмотрим процесс создания и настройки консольного приложения Win32 на языке C++ в среде Microsoft Visual Studio Community 2022.

#### **Создание решения**

После запуска среды на экране появляется ее главное окно, которое содержит заголовок, меню, панель инструментов, строку состояния, а также несколько дочерних окон, таких как *Обозреватель решений*, расположенный по умолчанию с правой стороны окна (рис. 3.1). В зависимости от настроек среды и пожеланий пользователя расположение и состав дочерних окон может отличаться от представленного на рисунке варианта. Также автоматически может запускаться начальное окно (рис. 3.1), с помощью которой можно

быстро выполнить одно из предложенных действий, например создание нового проекта или открытие уже существующего.

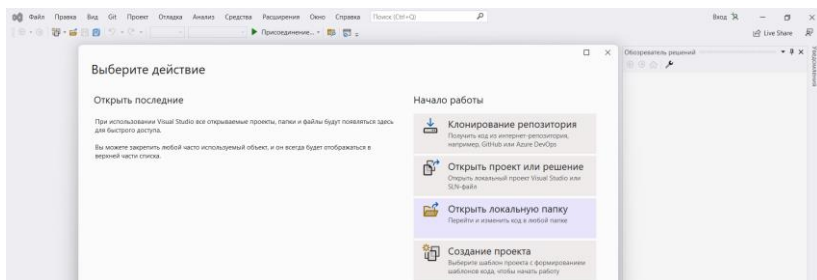


Рис. 3.1 Окно среды разработки программ Microsoft Visual Studio Community 2022

Если открытие начального окна при каждом запуске не требуется, его можно отключить в настройках среды (*Средства*→*Параметры*→*Окружение*→*Общие*).

Для создания консольного приложения можно щелкнуть по ссылке *Создание проекта* начального окна или выполнить команду меню *Файл*→*Создать*→*Проект*. На экране появится диалоговое окно (Рис. 3.2), в котором можно выбрать один из предлагаемых шаблонов для будущего приложения.

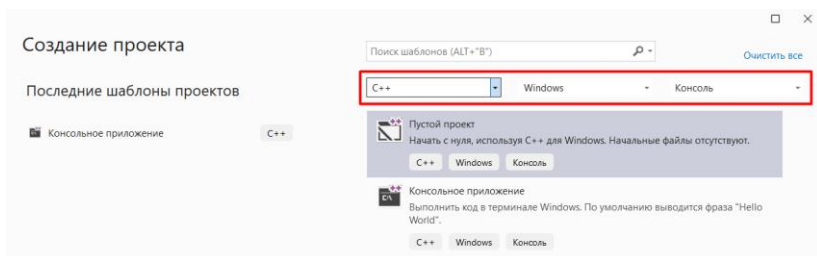


Рис. 3.2 Диалоговое окно *Создание проекта*

Для отбора нужного шаблона можно воспользоваться тремя полями с выпадающими списками в правой верхней части окна (Рис. 3.2). Один из них позволяет задать язык программирования (C++), второй

операционную систему (*Windows*), третий – тип приложения (*Консоль*). В перечне ниже располагаются отображенные в соответствии с этими параметрами варианты шаблонов. Выберем шаблон *Пустой проект* и нажмем кнопку *Далее*.

В следующем окне (рис. 3.3) в поле *Имя проекта* необходимо ввести имя нового проекта, в составе которого можно использовать как латиницу, так и кириллицу (например, «мой проект»). Одновременно автоматически заполняется поле *Имя решения*. Поле *Расположение* показывает место сохранения проекта, при желании его можно изменить. Флажок *Поместить решение и проект в одном каталоге* устанавливается в зависимости от предпочтений программиста в вопросах работы с файлами проекта, в данном случае устанавливать его не будем. После ввода имени проекта нажмем кнопку *Создать*. В результате будет создан проект с именем *мой проект*.

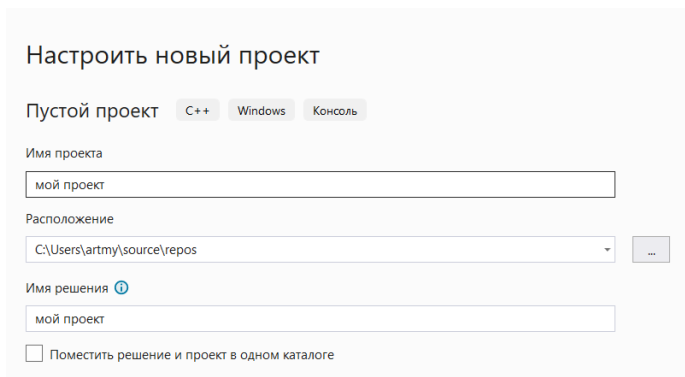


Рис. 3.3 Настройка нового проекта

В среде Visual Studio проект всегда создается в составе *решения*, которое может содержать несколько проектов (в нашем случае пока один). В свою очередь проект может объединять несколько логически связанных файлов с исходным кодом, настройками конфигурации,



ресурсами и т.д. Если перейти в папку, содержащую созданный проект, то можно увидеть там файл *мой проект.sln* – файл решения для созданной программы. Двойной щелчок на этом файле приведет к запуску среды с открытием данного решения.

Также в составе каталога с решением имеется подкаталог проекта, который также называется *мой проект* и содержит такие файлы:

- *мой проект.vcxproj* – файл проекта. Запуск этого файла также приведет к старту среды и открытию решения.
- *мой проект.vcxproj.filters* – файл с описанием фильтров, используемых *Обозревателем решения* для отображения разных групп файлов решения.
- *мой проект.vcxproj.user* – файл пользовательских настроек.

В дальнейшем в этом же каталоге будет размещаться файл с исходным текстом программы. Кроме того, в папке с решением будут созданы другие подкаталоги, о назначении которых будет рассказано ниже.

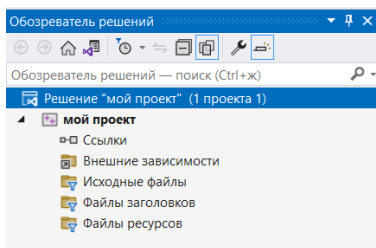


Рис. 3.4 Структура решения

Состав созданного в виде пустого проекта решения отображается в *Обозревателе решения* (Рис. 3.4) в виде набора папок. Папка *Внешние зависимости* отображает файлы, не добавленные явно в проект, но использующиеся в файлах исходного кода, например, включенные при помощи директивы *#include*. Остальные папки содержат,

соответственно, имеющиеся в проекте файлы исходного кода, заголовочные файлы и файлы ресурсов.

Созданный проект пока не содержит ни одного файла с исходным кодом. Для его добавления можно щелкнуть правой кнопкой мыши на папке *Исходные файлы* и выбрать команды *Добавить*, *Создать элемент* (Рис. 3.5).

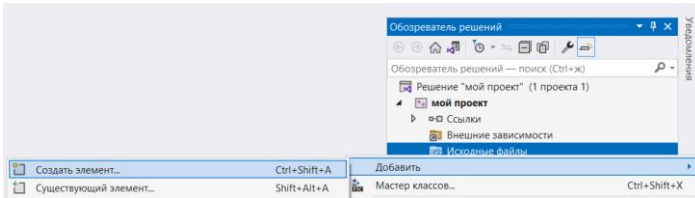


Рис. 3.5 Добавление в проект файла с исходным кодом

В диалоговом окне (Рис. 3.6) выберем тип элемента – *Файл C++ (.cpp)* и в поле *Имя* введем название будущего файла, например *main*. Если не указывать расширение, то автоматически к имени файла добавится *.cpp*.

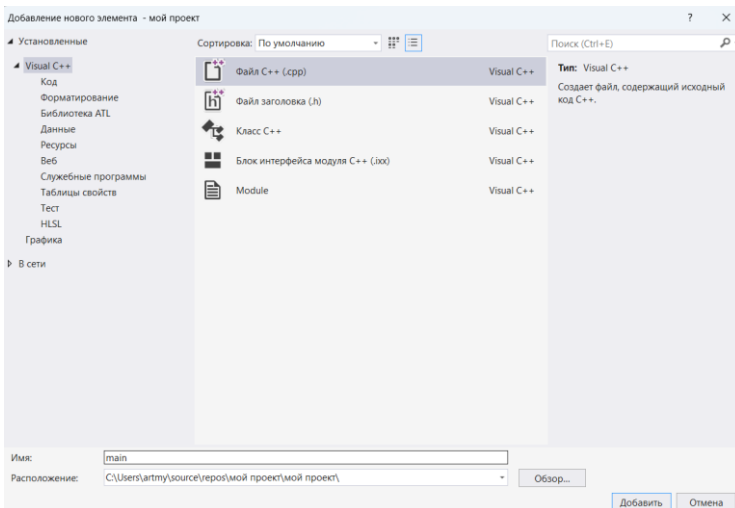


Рис. 3.6 Диалоговое окно Добавление нового элемента

После нажатия кнопки *Добавить* в окне редактора кода появляется новая вкладка файла *main.cpp* (**Ошибка! Источник ссылки не найден.**).

В дальнейшем в зависимости от действий пользователя и режимов работы среды в редакторе могут открываться в отдельных вкладках и другие файлы. Закрыть вкладку с файлом можно щелкнув по кнопке закрытия на ярлыке вкладки. Для открытия в редакторе кода какого-либо файла проекта достаточно дважды щелкнуть на нем в окне *Обозревателя решений*.

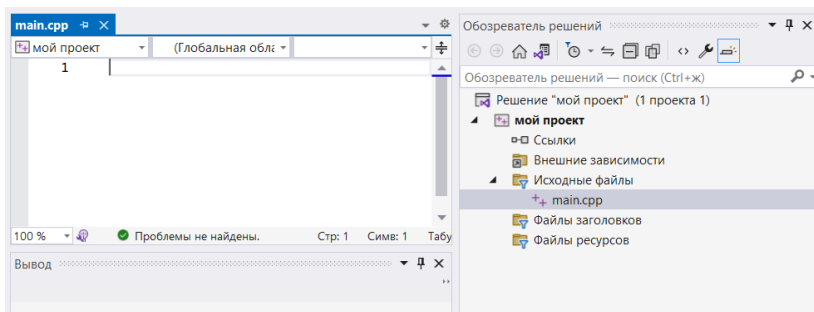


Рис. 3.7 Окно редактора с открытой вкладкой файла исходного кода

В состав решения можно добавлять и новые проекты. Для этого можно щелкнуть в *Обозревателе решений* правой кнопкой на строке с названием решения и выбрать команды *Добавить*, *Создать проект*. Дальнейшие действия с созданным проектом аналогичны тем, что рассматривались ранее. Какой из проектов будет автоматически запускаться при запуске решения определяется его настройками. Это может быть один из проектов или одновременно все. Для выбора нужного варианта требуется щелкнуть правой кнопкой мыши на строке с именем решения в обозревателе и выполнить команду *Свойства*. В окне настройки свойств решения можно настроить параметры запуска проектов.

Если необходимо прекратить работу с текущим решением, то можно выполнить команду *Файл→Закреть решение* или открыть другое решение командой *Файл→Открыть→Решение или проект*.

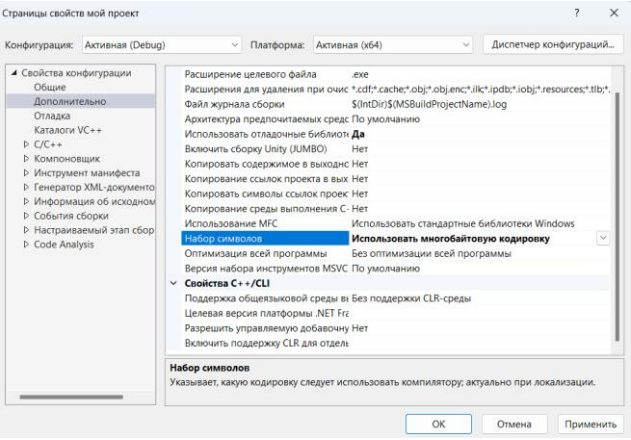


Рис. 3.8 Диалоговое окно определения свойств проекта

Теперь произведем настройку свойств созданного проекта. Для этого нужно выполнить команду меню *Проект→Свойства*. В левой части диалогового окна (**Ошибка! Источник ссылки не найден.**) раскроем узел *Свойства конфигурации*. Конфигурации проекта определяют параметры компоновки приложения и свойства устанавливаются для каждой из них отдельно. Изначально каждый проект в решении Visual Studio имеет две конфигурации — *Debug* (Отладка) и *Release* (Выпуск). Каждая из них создается (автоматически) в отдельном каталоге. При использовании конфигурации *Debug* будет создаваться отладочная версия проекта, с помощью которой можно осуществлять отладку на уровне исходного кода. Конфигурация *Release* предназначена для окончательной сборки приложения. В ней отсутствует отладочная информация и при создании выходного файла (с расширением .exe) компиляция происходит с включенным режимом

оптимизации кода, что уменьшает его объем (по сравнению с конфигурацией *Debug*). Для примера настроим некоторые свойства активной конфигурации *Debug*.

Кроме конфигурации можно изменять и целевую платформу, для которой производится компиляция исходного кода. Создаваться может 64-разрядное приложение (вариант *x64*) или 32-разрядное (вариант *x86*). Установить текущий вариант платформы и конфигурации можно с помощью полей на панели инструментов (рис. 3.9). А в окне свойств проекта (рис. 3.8.) можно индивидуально настраивать каждое сочетание конфигурации и платформы, и, кроме того, установить те параметры, которые являются общими для всех вариантов.

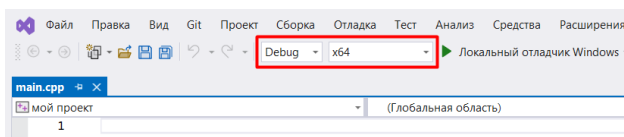


Рис. 3.9 Выбор конфигурации и целевой платформы проекта

После раскрытия узла *Свойства конфигурации* выберем пункт *Дополнительно* (рис. 3.8). Установим свойство *Набор символов* в значение *Использовать многобайтовую кодировку*. Данное свойство означает, что по умолчанию в программе будут использоваться строки и строковые константы в однобайтовой кодировке *ANSI* и соответствующие функции их обработки.

У проекта и его конфигураций имеется много других свойств. Например, в разделе *Общие* можно установить версию стандартов языков *C* и *C++*, в рамках которых будет работать компилятор. В разделе *Отладка* можно задать аргументы командной строки, если предполагается, что через нее приложение будет получать входные данные. Однако мы оставим эти свойства без изменения и нажмем *ОК*.

## Редактор кода

Среда Visual Studio включает в себя удобный редактор для набора исходного текста программы. Перечислим некоторые полезные возможности редактора кода:

- подсветка служебных слов языка и комментариев;
- разбиение кода на логические группы. Редактор автоматически объединяет фрагменты кода в группы, которые можно свернуть или развернуть, воспользовавшись значками с символами «-» и «+» слева от кода (Рис. 3.1010). Примером таких логических групп кода может быть блок комментариев или составной оператор тела функции;
- автоматическое создание отступов в коде. Для улучшения читабельности текста программы редактор автоматически добавляет отступы при вводе содержимого составного и некоторых других операторов. Также автоматически устанавливаются позиции самих открывающей и закрывающей фигурных скобок (с одинаковым отступом);
- удобное перемещение между началом и концом составного оператора. Если курсор расположен на открывающей фигурной скобке, то после нажатия **Ctrl+]** он автоматически перемещается на закрывающую скобку;
- демонстрация отладочной информации: выделение в тексте ошибок, размещение точек прерывания, закладок и других элементов отладки;
- поиск и замена фрагментов кода;
- вывод информации об определении элементов кода;
- использование технологии автодополнения *IntelliSense*.

Рассмотрим более подробно две последние возможности.

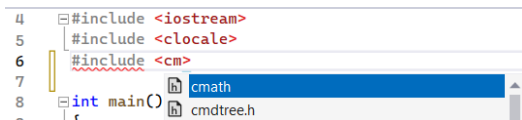


Рис. 3.10 Пример применения технологии *IntelliSense*

При работе с текстами программ (в особенности больших и состоящих из нескольких модулей) часто требуется получить напоминание о типе той или иной переменной или о том, как выглядит прототип (заголовок) какой-либо функции. Самый простой способ получения такой информации – подвести курсор мыши к нужному элементу и дождаться появления всплывающей подсказки. Если же требуется информация о том, в каком файле, и в каком месте этого файла определен элемент программы, то можно воспользоваться вкладкой *Окно определения кода* информационного окна (меню Вид→Окно определения кода), располагающегося при открытии в нижней части окна среды программирования. Если установить курсор на элементе программы, то через несколько секунд в *Окне определения кода* появится текст файла и будет выделена строка, содержащая нужное определение (Рис. 3.11).

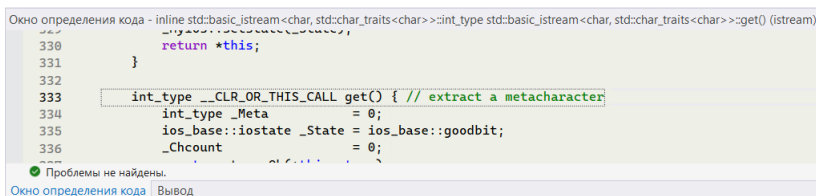


Рис. 3.11 Окно определения кода

По-другому отобразить определение элемента можно, если установить на него курсор и нажать комбинацию клавиш **Ctrl+F12**. Если определение находится в этом же файле, то оно будет выделено. Если в

другом, то он будет открыт в отдельной вкладке редактора кода и название элемента будет также выделено.

Технология автодополнения кода *IntelliSense* позволяет программисту не вводить полностью название функции или заголовочного файла, а выбрать их из списка, сформированного по первым введенным буквам. Например, если при наборе директивы препроцессора ввести `#include <`, то раскроется список всех доступных заголовочных файлов (рис. 3.10). Если вводить первые символы имени, то курсор начнет перемещаться по списку, сужая круг возможных вариантов. Для ввода выбранного варианта следует нажать **Tab** или **Enter**.

Для того, чтобы полностью не вводить название какой-либо функции или символической константы можно ввести первые несколько символов имени и будет показан список возможных вариантов (если список не отобразился, можно нажать **Ctrl+Пробел** для его принудительного отображения). Если нужная функция не найдена, то это значит, что в программу не был включен требуемый заголовочный файл. В этом случае необходимо добавить его с помощью директивы препроцессора `#include` и после этого повторить попытку ввода функции. После выбора нужной функции и ввода открывающей круглой скобки появится всплывающая подсказка о ее параметрах.

## Средства ввода-вывода данных

Прежде, чем рассмотреть примеры простейших программ, вкратце остановимся на тех средствах ввода-вывода, которые можно использовать в консольных приложениях.

Языки C и C++ лишены встроенных средств ввода-вывода, который осуществляется с помощью библиотечных функций. Языку C++ в



наследство от языка С досталась стандартная библиотека ввода-вывода, прототипы функций которой содержатся в заголовочном файле *cstdio* (аналог применяющегося в языке С заголовочного файла *stdio.h*). В стандартной библиотеке языка С++ также имеются развитые средства консольного ввода-вывода, объявления которых приведены в заголовочном файле *iostream*. Эти средства разработаны уже в парадигме объектно-ориентированного программирования. Для их использования в начале текста программы размещаются следующие инструкции:

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

Все идентификаторы стандартной библиотеки определены в пространстве имен *std*. Такой подход позволяет использовать одни и те же имена в разных библиотеках. Для доступа к таким объектам необходимо применять *уточненные* имена, указывая перед ними пространство имен и операцию разрешения области “::” (например, *std::cout*) или предварительно размещать директиву *using*. Во втором случае ниже строки с директивой можно использовать *не уточненные* имена объектов. Поэтому, как правило, если применяется директива *using*, то ее размещают в начале программы.

Заголовочный файл *iostream* содержит описание классов для управления вводом и выводом данных, а также определения стандартных объектов-потокв ввода с клавиатуры (*cin*) и вывода на экран (*cout*). Также в файле определены (перегружены) для разных типов данных операции помещения в поток << и чтения из потока >>. Это позволяет использовать одни и те же операции для переменных и выражений различных типов и не задумываться о каком-либо способе указания на принадлежность к таким типам. Кроме того, такие же

операции могут использоваться и при работе с файловыми потоками. Это возможно потому, что классы файловых потоков (будут рассматриваться в последующих работах) наследуют классам *std::istream* и *std::ostream*, которым, собственно, и принадлежат объекты-потоки *cin* и *cout* соответственно. Для примера их использования рассмотрим фрагмент, осуществляющий ввод с клавиатуры значений двух целых переменных и вывод их суммы на экран (в консоль):

```
int x,y;
std::cout<<"Введите два числа:"<<'\\n';
std::cin>>x>>y;
std::cout<<"Сумма чисел= "<<x+y<<std::endl;
```

При первом использовании объекта *cout* в консоль (а точнее в стандартный поток вывода, с которым связана переменная *cout*) выводится (помещается операцией <<) строковая константа *"Введите два числа:"*. Результатом применения операции помещения в поток является ссылка на тот же объект *cout*. Благодаря этому можно формировать цепочки операций, не указывая каждый раз имя объекта. И далее, после вывода строковой константы, в выходной поток помещается управляющая символьная константа *'\\n'*, в результате чего в окне консоли произойдет переход к началу новой строки.

Для ввода значений переменных *x* и *y* применяется объект-поток *cin* и операция извлечения из потока >>. Данная операция также позволяет осуществлять каскадное применение, так как аналогично возвращает ссылку на переменную *cin*. Затем с помощью объекта *cout* выводится строковая константа *"Сумма чисел= "*, а за ней результат вычисления выражения *x+y*. Завершает цепочку операций помещение в поток

манипулятора *endl* (определен в файле *iostream*), что приводит к тому же результату, что и помещение константы `'\n'`.

Из приведенного фрагмента можно сделать два вывода:

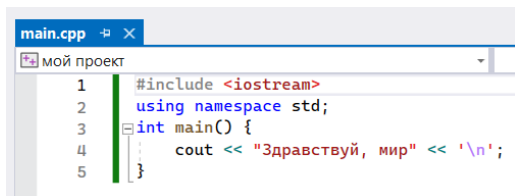
- перегрузка операций помещения в поток и извлечения из потока приводит к тому, что выражения и переменные различных типов обрабатываются ими автоматически без дополнительного указания формата данных;
- при выводе можно использовать два альтернативных варианта перевода строки: вывод управляющей символьной константы `'\n'` или применение манипулятора *endl*. Оба варианта равнозначны и выбор между ними ограничивается только предпочтениями программиста.

## Пример составления и отладки простейшей программы

Для примера в созданном ранее файле *main.cpp* разместим программу, которая выводит на экран фразу *Здравствуй, мир* (Рис. 3.122). Первая строка программы содержит директиву препроцессора, включающую в текст программы содержимое заголовочного файла *iostream*. Следом размещена директива *using*, предоставляющая доступ к стандартному пространству имен без необходимости использовать уточненные имена объектов. Далее идет определение функции *main*, которая обязательно должна быть в программе и с которой начинается ее выполнение (определение функции включает в себя заголовок и составной оператор или блок, содержащий ее тело). Согласно стандарта C++ функция *main* должна иметь тип *int* и возвращать системе значения нуля в случае успешного завершения с помощью оператора:

```
return 0;
```

Фактически же достижение при выполнении программы закрывающей фигурной скобки тела функции *main* эквивалентно применению данного оператора, поэтому его можно не указывать.

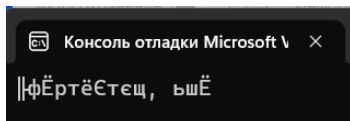


```
main.cpp
мой проект
1 #include <iostream>
2 using namespace std;
3 int main() {
4     cout << "Здравствуй, мир" << '\n';
5 }
```

Рис. 3.12 Пример программы

Для того, чтобы запустить набранную программу на выполнение можно использовать команду меню *Отладка*→*Начать отладку* или нажать **F5**. После начала компиляции сведения о ходе построения можно увидеть во вкладке *Вывод* информационного окна. Если в тексте будут обнаружены синтаксические ошибки, то в отдельную вкладку этого окна о них выводятся сообщения с указанием номера. Двойной щелчок мышью по сообщению перемещает курсор в место, где ошибка обнаружена.

В нашем случае ошибок нет и произойдет запуск программы в конфигурации, установленной по умолчанию (*Debug*). Откроется окно консоли отладки, где можно увидеть результаты работы программы (Рис. 3.133).



Консоль отладки Microsoft V

||фЁртёСтёщ, ьШЁ

Рис. 3.13 Результат работы программы

Однако в окне консоли мы не увидим слов *Здравствуй, мир*. Это происходит потому, что текст программы был набран в кодировке win-1251 (кодировка CP1251), а по умолчанию в окне консоли

используется кодовая страница CP866. Для ее изменения используем функцию *setlocale*, которая меняет так называемую схему локализации или *локаль*. Локаль определяет кроме кодировки символы валюты, систему мер и другие параметры. Прототип функции определен в файле *clocale*. У нее два параметра. Первый определяет локализуемую категорию. В данном случае это кодировка символов, поэтому укажем константу *LC\_CTYPE* (можно также *LC\_ALL* – все категории). В качестве второго параметра должна быть указана строка с названием локали. В нашем случае это *Russian* или *rus*. Однако поскольку программа будет выполняться под управлением русифицированной операционной системы, то можно записать в качестве названия пустую строку. Это означает, что будет применена локаль, используемая по умолчанию в данной системе. Таким образом, окончательный вариант программы выглядит так:

```
#include <iostream>
#include <clocale>
using namespace std;
int main() {
    setlocale(LC_CTYPE, "");
    cout << "Здравствуй, мир" << '\n';
}
```

Повторный запуск программы приведет к появлению в консоли искомой фразы.

Теперь рассмотрим такое средство отладки программ, как пошаговое выполнение. В Visual Studio имеется два варианта пошагового выполнения программы: с заходом внутрь функций (**F11**) и с обходом (**F10**). В данном примере другие функции, кроме *main*, не определены, поэтому можно использовать любую из этих команд. Для

примера нажмем **F10**. Далее при необходимости производится построение проекта и слева от первого оператора появляется желтая стрелка, показывающая текущий оператор. Также открывается окно консоли, где по мере выполнения программы будут появляться результаты ее работы. Далее последовательными нажатиями клавиши **F10** выполняем строки программы. В соответствующих вкладках информационного окна можно наблюдать значения переменных (при их наличии), информацию о вызовах библиотечных функций. Также в отдельной вкладке можно видеть стек вызовов функций, который удобно использовать при отладке программ с многими функциями, в том числе и при наличии рекурсивных функций.

Если после начала пошагового выполнения оказалось, что дальнейшей необходимости в нем нет, можно нажать комбинацию клавиш **Shift+F11** (шаг с выходом). Это приведет к выполнению оставшейся части программы в режиме обычной отладки. Другим вариантом является прекращение отладки (**Shift+F5**).

Еще одним средством отладки являются *точки останова*. Установив точку останова, можно запустить программу в режиме обычной отладки, а по достижении точки отладка перейдет в пошаговый режим. Для установки точки останова нужно кликнуть мышью в нужной строке текста в самой левой вертикальной полосе текущей вкладки текстового редактора. В результате в этой полосе появится красная точка (рис. 3.14), по достижении которой произойдет останов. Если подвести курсор мыши к точке, то появится всплывающее подменю, с помощью которого можно установить параметры ее срабатывания. Повторный клик по точке отключает ее.

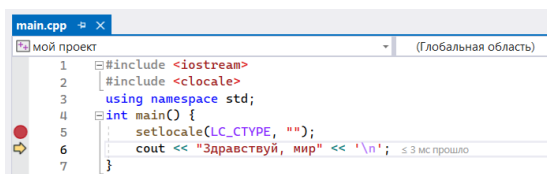


Рис. 3.14 Отладка программы с помощью точки останова

Если после запуска на отладку программы перейти в Проводнике в каталог решения, то можно увидеть, что кроме упомянутой ранее папки проекта дополнительно появилась папка с названием *x64*. Ее название соответствует выбранной ранее целевой платформе для компиляции программы. Внутри нее находится папка *Debug*, которая называется также, как выбранная ранее конфигурация проекта. В ней находится скомпонованный исполняемый файл *мой проект.exe*, который при желании можно запустить вне среды Visual Studio. Также в папке находится файл *мой проект.pdb*, содержащий отладочную информацию.

При выборе других комбинаций конфигурации и платформы в каталоге решения будут создаваться и другие папки. Например, если изменить конфигурацию на *Release* и повторно запустить отладку проекта, то в папке *x64* будет создана папка *Release*, где так же будет находиться исполняемый файл. Причем размер данного файла будет в несколько раз меньше, чем в конфигурации *Debug*, что связано с задействованием компилятором оптимизации кода.

## Условный оператор *if else*

Условный оператор применяется тогда, когда в программе нужно выбрать одно из двух альтернативных действий. Оператор имеет две формы: краткую (просто *if*) и полную (*if else*). Краткая форма имеет следующий синтаксис:

*if* (<выражение>)

<оператор>

Выражение в скобках должно иметь логический тип (*bool*). Выражения числовых типов (например, целых или вещественных) приводятся к логическому типу *bool*. Значение нуля – ложь (*false*), любое ненулевое значение – истина (*true*). Можно строить составные логические выражения, применяя логические операции И (&&), ИЛИ (||), НЕ (!). Если выражение в скобках истинно (т.е. отлично от нуля), то выполняется оператор (любой, в том числе и составной), расположенный после закрывающей круглой скобки. В противном случае управление переходит к следующему за условным оператором. На уровне блок-схемы это соответствует ситуации, когда в развилке ветвь *Нет* пустая.

Примеры:

```
int x=1,y=0,z;
```

```
//оператор z=5; будет выполнен, т.к. лог. выражение истинно:
```

```
if(y>=0)
```

```
    z=5;
```

```
//составной оператор не будет выполнен, т.к. лог. выражение ложно:
```

```
if(x<0 || y==1)
```

```
{
```

```
    y=20;
```

```
    z=30;
```

```
}
```

Оператор в полной форме (*if else*) позволяет определить, какой из двух операторов (или блоков – составных операторов) будет выполняться в зависимости от истинности или ложности проверяемого



логического выражения. Полная форма условного оператора имеет следующий синтаксис:

```
if (<выражение>)  
    <оператор1>  
else  
    <оператор2>
```

Если выражение в круглых скобках истинно (т.е. отлично от нуля), то выполняется *оператор1*. В противном случае – *оператор2*.  
Примеры:

```
int x=1,y=0,z;  
//переменная z будет равна 5:  
if(y!=5) z=5;  
else z=10;  
//в консоль будет выведено: z равно 10:  
if(x && y)  
{  
    y=20;  
    z=30;  
}  
else {  
    z=10;  
    std::cout<<"z равно "<<z<<'\\n';}
```

В качестве примера законченной программы, содержащей в своем составе условный оператор, рассмотрим решение следующей задачи:

*Составить программу для вычисления значения функции  $y = \frac{\ln x}{(x-3)}$  при заданном значении аргумента, вводимого с клавиатуры. Если введенное значение аргумента входит в область определения функции, то в консоль вывести значения аргумента и функции. В*

противном случае в консоль вывести сообщение, что вычисление функции при заданном значении аргумента невозможно.

Для заданной функции область определения будет ограничена следующими неравенствами:

$$\begin{cases} x > 0; \\ x \neq 3. \end{cases}$$

Программа для решения задачи будет иметь вид:

```
#include <iostream>
#include <locale>
#include <cmath>
using namespace std;
int main() {
    setlocale(LC_CTYPE, "");
    cout << "Введите значение x:";
    double x, y;
    cin >> x;
    if (x > 0 && x != 3) {
        y = log(x) / (x - 3);
        cout << "При x= " << x << " y= " << y << endl;
    }
    else
        cout << "Значение x= " << x << " вне ОДЗ" << endl;
}
```

Дополнительно к рассмотренным ранее заголовочным файлам, здесь в текст программы включается файл *cmath*, содержащий прототипы математических функций, так как была использована функция вычисления натурального логарифма *log*.

Условный оператор применяется в программе полной форме. В случае истинности выражения в скобках выполняется составной оператор (блок), в противном случае осуществляется вывод сообщения

о невозможности вычисления функции при заданном значении аргумента.

## СОДЕРЖАНИЕ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. В среде Visual Studio создать решение (консольное приложение). Настроить его свойства по аналогии с примером, рассмотренным в теоретических сведениях. В составе проекта решения добавить файл исходного кода на языке C++, в который ввести программу, выводящую в консоль ФИО студента, выполняющего работу и номер группы. Также программа должна содержать описание двух целочисленных переменных, которые вводятся с клавиатуры, а затем их сумма выводится в консоль.
3. При наборе программы отработать использование основных возможностей редактора кода.
4. Произвести отладку программы в обычном и пошаговом режимах.
5. В отчет внести текст программы, а также скриншоты информационного окна после построения и при пошаговом выполнении программы (со значениями локальных переменных) и окна консоли с результатом работы программы.
6. Выбрать алгоритм, составить его блок-схему и программу на языке C++ с использованием условного оператора *if* для вычисления и вывода в консоль при заданном значении  $x$  значения функции  $y=f(x)$ , указанной в варианте задания (см. ниже). В случае, если введенное значение  $x$  не принадлежит области определения функции, выводить информационное сообщение.

7. В созданном ранее решении добавить еще один проект, в состав которого ввести файл с исходным кодом составленной программы. Свойства созданного проекта настроить аналогично предыдущему проекту. В свойствах решения установить запуск на отладку одновременно обоих проектов.
8. Осуществить отладку и тестирование программы, подготовив не менее двух тестовых значений  $x$ , входящих в область определения функции и двух не входящих в нее. Убедиться, что программа при вводе тестовых значений работает корректно. Сравнить результаты вычислений функции при тестовых значениях  $x$  с заранее вычисленными с помощью MS Excel при тех же значениях аргумента.
9. В отчет внести текст программы, тестовые значения аргумента, проверочную формулу MS Excel и результаты вычислений с ее помощью, скриншоты консоли с результатами тестовых запусков.

## ВАРИАНТЫ ЗАДАНИЯ

1.  $y = \frac{x}{x^2 - 1} + \log_3(x + 2),$
2.  $y = \frac{x^3}{(x+1)(x+2)} + \frac{\arcsin(1-x)}{\sqrt[3]{1-\ln x}},$
3.  $y = \frac{\sin x}{1 - \cos x} \cdot \frac{\operatorname{tg}^3(\ln(1-x))}{|1 + x \cdot e^{-x}|},$
4.  $y = \frac{-\arccos(1-x)}{\sqrt[4]{x^3 - 1}} + (2-x) \cos^2|x|,$
5.  $y = \frac{\cos^2 x}{1 + \sin x} - \ln^2\left(\frac{x}{\sqrt[3]{x-1}}\right),$
6.  $y = \frac{x^3 e^{x-1}}{x^3 - |x|} - \log_2(\sqrt{x} - x),$

7.

$$y = \frac{\sqrt[3]{x + \sin x}}{x^2 - x^4} \cdot \arcsin^2 \sqrt[4]{3 - x},$$

9.

$$y = \frac{\sin x + \frac{1}{x}}{\sqrt[3]{\operatorname{tg}^2 \left( -\frac{x^3}{x^2 - 4} \right)}} + 2^{|x-1|},$$

11.

$$y = \frac{\sqrt[4]{8x^2 - 6x + 1}}{\operatorname{arctg} \sqrt{2x + 1}} + 2^{\sin x / |x|},$$

13.

$$y = \sqrt[3]{\log_2(1 - x)} + \frac{\operatorname{tg}(1 + 1/x)}{\sqrt{|x| - 2}},$$

15.

$$y = \frac{\sqrt[4]{|x|} + 1}{\sin^2 \frac{x}{2} - 1} + 2^{\sqrt{x+1}},$$

17.

$$y = \frac{x^3 + \sin(3|x| - 1)}{1 - \cos^2 x} - \log_2(3^x - 9),$$

19.

$$y = (1 + x)^{\sin \sqrt{x}} \cdot 2^{\cos^2 \left( \frac{x}{x-2} \right)},$$

21.

$$y = 2^{|x|} \cdot \ln |\sin x^4| - \cos^2 \sqrt{4 - x^2},$$

$$8. \quad y = \frac{x^5 + e^{-2|x|}}{\sqrt[4]{9 - x^2}} \cdot \operatorname{tg}^3 |\cos^2 x|,$$

$$10. \quad y = \frac{\sin^2 \frac{|x|}{2} + 3^{\frac{1}{x-1}}}{\sqrt[6]{x^4 - 16}} \cdot \sqrt{1 - \ln x},$$

12.

$$y = \frac{\ln^3 \frac{(x-1)^2}{x} + \cos^2(2x)}{\sqrt[6]{x^2 - 5x + 6}} \cdot \sin \frac{3x^2 - 1}{2},$$

14.

$$y = \frac{1}{x} \log_3(4 - x^2) + \frac{\sin(\cos x)}{e^{|x|} - 1},$$

16.

$$y = \sqrt{\frac{1}{x}(x^2 - 1)} \cdot \cos^2 \frac{|x|}{3} + \lg \frac{1}{x+1},$$

18.

$$y = \frac{\sin^2 \sqrt[3]{x}}{x} + e^{-\sqrt{x^2 - 6x + 8}},$$

20.

$$y = \frac{-x^2}{(2x+2)(2x-3)} + \frac{\log_2(\sqrt{x} - 1)}{\sin 2x},$$

22.

$$y = \left[ \cos \left( e^{\sqrt{|x|-2}} + x^3 \right) \right]^{2x} - \frac{|x|}{x - \sqrt{x}},$$

$$23. \quad y = e^{x^2-1} + \frac{x \cdot \sin \frac{1}{x}}{\sqrt[4]{9-\sqrt{x}}},$$

24.

$$y = \frac{x}{\cos(x-\pi/2)\sin^2(x-\pi/2)} + e^{\sqrt{x-|x|}},$$

25.

$$y = \frac{2^{x^2} + \sqrt{16-x^2}}{\sqrt[3]{x-2}} + \operatorname{tg}^2\left(\frac{x}{x+2}\right),$$

$$26. \quad y = \frac{1 + \log_2(\sin 2x)}{1-2x} + \frac{\sqrt[2]{|x|-1}}{x^3-27},$$

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что в среде Visual Studio называется решением? Чем решение отличается от проекта?
2. Как создать решение?
3. Как в готовое решение добавляется еще один проект?
4. Как в состав проекта добавляются новые файлы?
5. Какие типы файлов входят в состав решение, проекта?
6. Каким образом в среде Visual Studio настраиваются свойства проекта?
7. Перечислите основные возможности редактора кода в среде Visual Studio?
8. Что такое конфигурация проекта? Как ее можно изменить?
9. Как можно произвести локализацию создаваемого консольного приложения?
10. Каким образом в среде Visual Studio производится отладка создаваемого приложения? Какие средства отладки вы знаете?
11. Каков синтаксис условного оператора языка C++? В каких формах он может применяться?
12. Каким может быть тип проверяемого выражения в составе условного оператора?

## Лабораторная работа №4

# РЕАЛИЗАЦИЯ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ СРЕДСТВАМИ ЯЗЫКА C++

**Цель работы:** получить навыки в составлении простейших циклических алгоритмов и реализации их средствами языка C++.

## КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

### Операторы цикла языка C++

В языке C++ имеются три оператора цикла: *for*, *while*, *do-while*. Далее будет более подробно рассмотрен каждый из этих операторов.

### Оператор цикла *for*

Данный оператор используется в основном в тех случаях, когда в программе необходимо организовать выполнение цикла с параметром, т.е. цикла в котором некоторая переменная изменяется от начального до конечного значения с заданным шагом. Синтаксис этого оператора следующий:

*for* (<выражение\_1>; <выражение\_2>; <выражение\_3>)  
    <оператор>

Тело цикла составляет либо один оператор, либо несколько операторов, заключенных в фигурные скобки (составной оператор).

*Выражение\_1* описывает инициализацию цикла (обычно присваивает начальное значение управляющей переменной).

*Выражение\_2* – проверка условия завершения цикла. Если оно истинно, то выполняется оператор тела цикла.

*Выражение\_3* вычисляется после каждой итерации (обычно изменяет на каждом шаге значение управляющей переменной).

Пример:

```
for (int i=1; i<=10; i++)
    std::cout<<"i= "<<i<<'\\n';
```

Этот фрагмент программы осуществляет вывод на экран десяти значений переменной  $i$  от 1 до 10.

В цикле *for* параметр может иметь любой тип и изменяться с произвольным шагом.

Любое из трех выражений может отсутствовать, но точки с запятой их разделяющие опускать нельзя

```
for ( ; ; )// Бесконечный цикл
```

Выражения 1 и 3 могут состоять из нескольких выражений, объединенных операцией запятой.

Пример:

*Написать программу, вычисляющую значение функции  $z = \ln(x)/\sin(y)$  при  $x \in [1;1.5]$  изменяющимся с шагом  $h_1 = 0,1$  и  $y \in [1;2]$  и изменяющимся с шагом  $h_2 = 0,2$ .*

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double x,y,z;
    for (x=1,y=1; y<=2; x=x+0.1, y=y+0.2)
    {
        z=log(x)/sin(y);
        cout<<"x= "<<x<<" y= "<<y<<" z= "<<z<<endl;
    }
}
```



## Оператор цикла *while*

Данный оператор реализует цикл с предусловием. Его синтаксис:

```
while (<выражение>)  
    <оператор>
```

Если выражение в скобках истинно, то выполняется оператор тела цикла, который может быть и составным.

Пример:

*Реализовать предыдущий пример, с применением оператора *while*.*

```
#include <iostream>  
#include <cmath>  
using namespace std;  
int main()  
{  
    double x=1,y=1,z;  
    while (y<=2)  
    {  
        z=log(x)/sin(y);  
        cout<<"x= "<<x<<" y= "<<y<<" z= "<<z<<endl;  
        x+=0.1;  
        y+=0.2;  
    }  
}
```

## Оператор цикла *do-while*

Данный оператор реализует цикл с постусловием. Его синтаксис:

```
do  
    <оператор>  
while (<выражение>);
```

Если выражение в скобках истинно, то выполняется оператор тела цикла.

Пример:

*Реализовать предыдущий пример, с применением оператора do-while.*

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double x=1,y=1,z;
    do
    {
        z=log(x)/sin(y);
        cout<<"x= "<<x<<" y= "<<y<<" z= "<<z<<endl;
        x+=0.1;
        y+=0.2;
    }
    while (y<=2);
}
```

## СОДЕРЖАНИЕ РАБОТЫ

1. Выбрать алгоритм, составить его блок-схему и программу с использованием оператора цикла *for* для вычисления и вывода в консоль в точках  $x_i = a + i \cdot h$ ,  $i=0, 1, 2, \dots, n$ ,  $h=(b-a)/n$  промежутка  $[a, b]$  значений функции  $y=f(x)$ , указанной в варианте задания (см. ниже). Также программа должна определять наибольшее и среднее значение функции. Предусмотреть проверку вычисляемых значений аргумента на принадлежность области

определения функции. Ввод исходных данных ( $a$ ,  $b$ ,  $n$ ) осуществлять с клавиатуры.

2. Составить аналогичные блок-схему и программу, но с использованием оператора цикла *while* или *do-while* на выбор.
3. Создать новое решение, в которое в виде отдельных проектов включить программы, созданные при выполнении пунктов 1 и 2. В отчет внести обе блок-схемы и программы, а также результаты их тестирования.

### ВАРИАНТЫ ЗАДАНИЯ

$$1. \quad y = \frac{x}{x^2 - 1} + \log_3(x + 2),$$

$$x \in [2; 3], n = 10$$

$$2. \quad y = \frac{x^3}{(x+1)(x+2)} + \frac{\arcsin(1-x)}{\sqrt[3]{1-\ln x}},$$

$$x \in [1; 2], n = 10$$

$$3. \quad y = \frac{\sin x}{1 - \cos x} \cdot \frac{\operatorname{tg}^3(\ln(1-x))}{|1 + x \cdot e^{-x}|},$$

$$x \in [-1; -0,5], n = 5$$

4.

$$y = \frac{-\arccos(1-x)}{\sqrt[4]{x^3 - 1}} + (2-x) \cos^2 |x|,$$

$$x \in [1,5; 2], n = 5$$

$$5. \quad y = \frac{\cos^2 x}{1 + \sin x} - \ln^2 \left( \frac{x}{\sqrt[3]{x-1}} \right),$$

$$x \in [2; 3], n = 10$$

$$6. \quad y = \frac{x^3 e^{x-1}}{x^3 - |x|} - \log_2(\sqrt{x} - x),$$

$$x \in [0,2; 0,8], n = 6$$

7.

$$y = \frac{\sqrt[3]{x + \sin x}}{x^2 - x^4} \cdot \arcsin^2 \sqrt[4]{3-x},$$

$$x \in [2; 3], n = 10$$

$$8. \quad y = \frac{x^5 + e^{-2|x|}}{\sqrt[4]{9-x^2}} \cdot \operatorname{tg}^3 |\cos^2 x|,$$

$$x \in [1; 2], n = 10$$

9.

$$y = \frac{\sin x + \frac{1}{x}}{\sqrt[3]{\operatorname{tg}^2\left(-\frac{x^3}{x^2-4}\right)}} + 2^{|x-1|},$$

$$x \in [1; 2], n = 5$$

11.

$$y = \frac{\sqrt[4]{8x^2-6x+1}}{\operatorname{arctg}\sqrt{2x+1}} + 2^{\sin x/|x|},$$

$$x \in [1; 2], n = 10$$

13.

$$y = \sqrt[3]{\log_2(1-x)} + \frac{\operatorname{tg}(1+1/x)}{\sqrt{|x|-2}},$$

$$x \in [-2; -1], n = 10$$

15.

$$y = \frac{\sqrt[4]{|x|+1}}{\sin^2 \frac{x}{2} - 1} + 2^{\sqrt{x+1}},$$

$$x \in [0; 1], n = 10$$

17.

$$y = \frac{x^3 + \sin(3|x|-1)}{1 - \cos^2 x} - \log_2(3^x - 9),$$

$$x \in [3; 4], n = 10$$

$$10. \quad y = \frac{\sin^2 \frac{|x|}{2} + 3^{\frac{1}{x-1}}}{\sqrt[6]{x^4-16}} \cdot \sqrt{1-\ln x},$$

$$x \in [2; 2, 6], n = 4$$

12.

$$y = \frac{\ln^3 \frac{(x-1)^2}{x} + \cos^2(2x)}{\sqrt[6]{x^2-5x+6}} \cdot \sin \frac{3^{x^2-1}}{2},$$

$$x \in [3; 5; 4], n = 5$$

$$14. \quad y = \frac{1}{x} \log_3(4-x^2) + \frac{\sin(\cos x)}{e^{|x|}-1},$$

$$x \in [0; 5; 1, 5], n = 10$$

16.

$$y = \sqrt{\frac{1}{x}(x^2-1)} \cdot \cos^2 \frac{|x|}{3} + \lg \frac{1}{x+1},$$

$$x \in [1; 2], n = 10$$

$$18. \quad y = \frac{\sin^2 \sqrt[3]{x}}{x} + e^{-\sqrt{x^2-6x+8}},$$

$$x \in [1; 2], n = 10$$

$$19. \quad y = (1+x)^{\sin \sqrt{x}} \cdot 2^{\cos^2 \left( \frac{x}{x-2} \right)},$$

$$x \in [0; 1], n = 10$$

20.

$$y = \frac{-x^2}{(2x+2)(2x-3)} + \frac{\log_2(\sqrt{x}-1)}{\sin 2x},$$

$$x \in [2; 3], n = 10$$

21.

$$y = 2^{|x|} \cdot \ln |\sin x^4| - \cos^2 \sqrt{4-x^2},$$

$$x \in [1; 2], n = 10$$

22.

$$y = \left[ \cos \left( e^{\sqrt{|x|-2}} + x^3 \right) \right]^{2x} - \frac{|x|}{x - \sqrt{x}},$$

$$x \in [2; 3], n = 10$$

$$23. \quad y = e^{x^2-1} + \frac{x \cdot \sin \frac{1}{x}}{\sqrt[4]{9-\sqrt{x}}},$$

$$x \in [1; 2], n = 10$$

24.

$$y = \frac{x}{\cos(x-\pi/2) \sin^2(x-\pi/2)} + e^{\sqrt{x}-|x|},$$

$$x \in [2; 3], n = 10$$

25.

$$y = \frac{2^{x^2} + \sqrt{16-x^2}}{\sqrt[3]{x-2}} + \operatorname{tg}^2 \left( \frac{x}{x+2} \right),$$

$$x \in [3; 4], n = 10$$

$$26. \quad y = \frac{1 + \log_2(\sin 2x)}{1-2x} + \frac{\sqrt[2]{|x|-1}}{x^3-27},$$

$$x \in [1; 1.5], n = 5$$

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каков формат записи оператора цикла *for*.
2. Каков формат записи оператора цикла *while*.
3. Каков формат записи оператора цикла *do-while*.

4. Каким образом можно включить несколько операторов в тело цикла?
5. Может ли управляющая переменная в цикле *for* быть вещественной?
6. Может ли управляющая переменная в цикле *for* изменяться в его теле?
7. Допустима ли форма записи цикла *for*, в которой отсутствует условие выхода? Если да, то сколько раз выполнится такой оператор?
8. При каких условиях оператор цикла *while* прекращает свое выполнение?
9. При каких условиях оператор цикла *do-while* прекращает свое выполнение?
10. Отличия оператора цикла *while* от *do-while*.

## Лабораторная работа №5

### ОБРАБОТКА ОДНОМЕРНЫХ МАССИВОВ

**Цель работы:** приобрести практический опыт использования одномерных массивов.

#### КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Массивы являются примером структурных (составных) типов данных. Массив – это индексированный набор переменных определенного типа, имеющий общее для всех своих элементов имя. В зависимости от количества индексов, определяющих положение элемента, массивы подразделяются на одномерные, двумерные и т.д. Обращение к элементам массива производится указанием имени и индекса в квадратных скобках. Пример использования одномерного массива:

```
#include <iostream>
#include <locale>
using namespace std;
const int N = 3;
int main()
{
    setlocale(LC_CTYPE, "");
    // Описание массива из трех элементов list:
    int list[N];
    // Присваивание элементам массива значений:
    list[0] = 421;
    list[1] = 53;
    list[2] = 1806;
    // Вывод элементов массива в консоль:
```

```

    cout<<"Элементы массива:"<<endl;
    for(int i=0; i<N; i++)
        cout<<i+1<<"-й элемент: "<<list[i]<<endl;
}

```

Результаты работы данной программы будут иметь вид:

Элементы массива:

1-й элемент: 421

2-й элемент: 53

3-й элемент: 1806

Выражение `int list[N]` объявляет `list` как массив переменных типа `int`, с объемом памяти, выделяемым для трех целых переменных (так как константа `N` равна 3). Индексы массива всегда целые и начинаются с нуля. К первой переменной массива можно обращаться как `list[0]`, ко второй – как `list[1]`, к третьей – как `list[2]`. В общем случае описание любого массива имеет следующий вид:

`<тип> <имя>[размер]`

Наряду с непосредственным присваиванием, существуют и другие способы получения элементов массива. Например, элементы массива можно ввести с клавиатуры:

```

int x[10];
std::cout<<"Введите 10 элементов массива:"<<'\\n';
for(int i=0; i<10; i++)
{
    std::cout<<i+1<<"-й элемент --> ";
    std::cin>>x[i];
}

```

Другим способом является непосредственная инициализация массива:

```
int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```



```
double b[5]={1.7e3, 5.23, 15};
```

```
char c[]={97, 'b', 'c', '\0'};
```

Если в фигурных скобках количество заданных значений меньше размерности массива (*b*), то оставшиеся элементы будут иметь нулевое значение. Пустые квадратные скобки после имени массива означают, что его размерность определяется по количеству заданных в фигурных скобках значений (массив *c* состоит из четырех элементов, по сути, инициализируется строковой константой “*abc*”).

Начиная со стандарта языка C++11, для переменных доступна так называемая списковая инициализация, когда символ “=” не указывается. Применительно к массивам это выглядит так:

```
int d[20]{20, 25};
```

Кроме того, что символ “=” не указывается, других отличий в правилах инициализации, по сравнению с первоначальным способом, не имеется. То есть первые два элемента массива *d* после такой инициализации равны 20 и 25 соответственно, а остальные равны нулю.

## СОДЕРЖАНИЕ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Выбрать алгоритм, составить его блок-схему и программу для решения выбранного варианта задания. Исходный массив может быть введен с клавиатуры или инициализирован при описании.
3. Исходные и результирующие массивы вывести в консоль в виде:

$x_0 \ x_1 \ x_2 \ x_3 \ x_4$

$x_5 \ x_6 \ x_7 \ x_8 \ x_9$

$x_{10} \ x_{11} \ x_{12} \ x_{13} \ x_{14}$

$x_{15} \ x_{16} \ x_{17} \ x_{18} \ x_{19}$

## ВАРИАНТЫ ЗАДАНИЯ

1. В заданном массиве  $X$ , состоящем из 20 элементов, определить и вывести на экран первый отрицательный элемент и его порядковый номер, а затем заменить его произведением предшествующих значений. Если все элементы положительны, выдать соответствующее сообщение.
2. Задан целочисленный массив  $X$  из 20 элементов. Вывести на экран все группы идущих подряд одинаковых элементов. Выдать соответствующее сообщение, если таких групп элементов в массиве нет.
3. Задан массив  $X$  из 20 элементов и число  $N$  ( $N < 20$ ). Не прибегая к сортировке, определить и вывести на экран  $N$  наибольших элементов массива.
4. В заданном целочисленном массиве  $X$ , состоящем из 20 элементов и упорядоченном по неубыванию, определить и вывести на экран те элементы, которые можно представить суммой двух других элементов.
5. Задан целочисленный массив  $X$  из 20 элементов. Определить и вывести на экран те элементы, делителем которых является хотя бы один из других элементов.
6. В заданном массиве  $X$ , состоящем из 20 элементов, определить и вывести на экран количество положительных, отрицательных и равных нулю элементов. Если положительных элементов больше (меньше), чем отрицательных, то заменить нулями нужное число положительных (отрицательных) элементов, чтобы их количество совпадало.

7. Задан целочисленный массив  $X$  из 20 элементов. Из этого массива переписать в массив  $Y$  ту последовательность, которая образует арифметическую прогрессию как минимум из пяти членов. Выдать соответствующее сообщение, если таких последовательностей нет.
8. Задан целочисленный массив  $X$  из 20 элементов. Переписать в массив  $Y$  те элементы исходного массива, которые строго больше двух своих соседей. Элементы массива  $Y$  не должны повторяться.
9. Задан целочисленный массив  $X$  из 20 элементов. Из этого массива переписать в массив  $Y$  той же размерности подряд все отрицательные элементы, а оставшиеся места заполнить единицами. Расположить элементы образованного массива в порядке убывания.
10. Задан целочисленный массив  $X$  из 20 элементов. Определить, можно ли из его положительных элементов составить строго возрастающую последовательность.
11. Задан целочисленный массив  $X$  из 20 элементов, содержащий как четные, так и нечетные числа. Из этого массива переписать в массив  $Y$  подряд первые пять различных четных элементов. Если таких элементов менее пяти, заполнить оставшиеся позиции в массиве суммой нечетных элементов массива  $X$ .
12. Задан целочисленный массив  $X$  из 20 элементов, содержащий группы подряд идущих одинаковых элементов. Поменять местами первую и последнюю группы массива.
13. Задан целочисленный массив  $X$  из 20 элементов. Определить максимальное количество идущих подряд и упорядоченных по возрастанию положительных чисел.

14. Задан целочисленный массив  $X$  из 20 элементов. Определить сумму элементов, имеющих четные индексы и являющихся нечетными числами. Если таковых нет, увеличить на единицу все элементы с четными индексами и вывести на экран результирующий массив.
15. Задан массив  $X$  натуральных чисел из 20 элементов. Удалить из него элементы, являющиеся удвоенными нечетными числами.
16. Задан массив  $X$  натуральных чисел из 20 элементов. Переписать в массив  $Y$  элементы, дающие при делении на 7 остаток 1, 2 или 5. Элементы массива  $Y$  упорядочить по неубыванию. В случае отсутствия таких элементов выдать соответствующее сообщение.
17. Задан целочисленный массив  $X$  из 20 элементов. Переписать в массив  $Y$  те элементы, которые равны сумме предшествующих им значений.
18. Задан действительный массив  $X$  из 20 элементов, содержащий 10 положительных и 10 отрицательных чисел. Переставить элементы массива так, чтобы положительные и отрицательные числа чередовались.
19. Задан целочисленный массив  $X$  из 20 элементов, среди которых есть повторяющиеся. Переписать в массив  $Y$  только неповторяющиеся элементы исходного массива.
20. Задан целочисленный массив  $X$  из 20 элементов, среди которых есть повторяющиеся. Записать в массив  $Y$  по одному элементу из каждой группы одинаковых значений исходного массива.
21. Задан целочисленный массив  $X$  из 20 элементов. Получить массив  $Y$ , в который переписать те положительные элементы массива  $X$ , которые расположены между двумя отрицательными.

Если таких элементов нет, вывести соответствующее сообщение.

Элементы массива  $Y$  не должны повторяться

22. Задан целочисленный массив  $X$  из 20 элементов. Переписать в массив  $Y$  наибольшую по длине возрастающую последовательность исходного массива.
23. Задан целочисленный массив  $X$  из 20 элементов, среди которых есть повторяющиеся. Определить наименьший и наибольший элементы массива. Если они встречаются несколько раз, то оставить их по одному экземпляру, заменив остальные вхождения средним арифметическим наибольшего и наименьшего элементов.
24. Задан целочисленный массив  $X$  из 20 элементов, содержащий как положительные, так и отрицательные значения. Переставить элементы в массиве так, чтобы в начале располагались все положительные элементы, а затем все отрицательные. Порядок следования элементов в этих группах должен остаться прежним.
25. Задан целочисленный массив  $X$  из 20 элементов. Получить массив  $Y$ , в который записать те из элементов исходного массива в порядке следования, которые образуют наиболее длинную возрастающую последовательность.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Дайте определение массива.
2. Как производится доступ к отдельным элементам массива?
3. Что такое указатель? Как он описывается?
4. Что общего у понятий массива и указателя?
5. Как с помощью указателя обратиться к элементу массива?
6. Какие способы заполнения массива значениями вы знаете?

7. Как определяется символьный массив?
8. Что представляет собой строка символов?
9. Каково внутреннее представление строковых констант?
10. Чем ограничен размер строки символов?
11. Какие ограничения накладываются на индексы массивов?
12. Может ли быть индекс массива равен значению его размерности?
13. Можно ли при описании массива не указывать его размерность?
14. Что такое списковая инициализация массива? В чем ее особенности?

## **Лабораторная работа №6**

### **ОБРАБОТКА ДВУМЕРНЫХ МАССИВОВ.**

### **ФАЙЛОВЫЙ ВВОД-ВЫВОД.**

**Цель работы:** ознакомиться с организацией двумерных массивов в языке C++, в том числе и динамических; приобрести практические навыки в файловом вводе-выводе данных.

### **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

#### **Организация многомерных массивов**

В языке C++ многомерные (в частности двумерные) массивы – это массивы, элементами которых в свою очередь являются другие массивы. Например, конструкция:

```
int mass[3][5];
```

описывает массив из трех элементов, каждым из которых является массив из пяти элементов целого типа. Фактически это матрица (3×5). Доступ к элементам осуществляется указанием двух индексов:

```
mass[0][1]=25; // присвоить 25 элементу, находящемуся в  
                // 1-й строке и 2-м столбце
```

Для работы с двумерными массивами, как правило, требуется применение вложенных циклов. В качестве примера рассмотрим ввод элементов массива с клавиатуры:

```
int a[5][5];  
for(int i=0; i<5; i++)  
    for(int j=0; j<5; j++)  
        std::cin>>a[i][j];
```

Двумерные массивы можно инициализировать при описании так же, как и одномерные. Например:

```
int a[][3]{
    {1,2,3},
    {4,5,6}};
```

### Динамические массивы

Для распределения динамической памяти в языке C++ имеется операция *new*. Например, для создания целочисленного динамического одномерного массива в общем случае из *n* элементов, можно использовать следующий фрагмент:

```
int *p, n=20;
p = new int[n];
```

Созданный динамический массив ничем неявно не инициализируется. Обращение к элементу такого массива производится вполне привычным образом, например:

```
p[15] = 25;
```

Также, как и для статических массивов, для динамических допустимо использовать списковую инициализацию, например:

```
int m = 10;
double *x = new double[m]{5.34, 12.03, 34.87};
```

Создание динамического двумерного массива с заранее неизвестными числом строк и столбцов проводится в два этапа. Сначала создается одномерный массив указателей, каждый из элементов которого будет ссылаться на соответствующую строку матрицы. Затем создаются сами строки (то есть одномерные массивы базового типа), указатели на которые размещаются в ранее созданном массиве указателей. Пример создания и ввода элементов двумерного динамического массива:

```
int **matr, n, m;
std::cout<<"Введите количество строк и столбцов:"<<"\n";
```



```

std::cin>>n>>m;
matr = new int *[n];
for(int i=0; i<n; i++)
{
    matr[i] = new int[m];
    for(int j=0; j<m; j++)
        std::cin>>matr[i][j];
}

```

Освобождение памяти, занимаемой массивом, вне зависимости от числа измерений, производится операцией *delete []*. Например:

```
delete []matr;
```

## **Потоковые классы в языке C++**

В языке C++ ввод-вывод осуществляется средствами стандартной библиотеки. Основным средством ввода-вывода является *поток* – абстрактное понятие, используемое для обмена данными между программой и источником информации или местом ее назначения. При вводе программа извлекает из входного потока байты данных, а при выводе – помещает их в выходной поток. Сами потоки могут ассоциироваться с самыми разными устройствами, но во всех случаях используются унифицированные механизмы обмена, скрывающие особенности функционирования таких средств от программиста.

Большинство используемых видов потоков являются *буферизированными*. Это означает, что при вводе порция данных считывается из устройства в специальный буфер в оперативной памяти, откуда уже может постепенно извлекаться прикладной программой по мере ее работы. После опустошения буфера в него вновь считывается порция данных и процесс повторяется. При выводе данные поступают сначала в буфер, а затем, при его заполнении или при выполнении

специальной инструкции, происходит сброс буфера и данные передаются устройству. В целом, буферизация ввода-вывода позволяет согласовывать скорости работы двух сторон в процессе обмена данными.

Для работы с потоками стандартная библиотека содержит иерархию классов, в основе которой находятся два базовых класса: *ios* и *streambuf*. Первый содержит общие поля и методы, второй – добавляет механизм буферизации ввода-вывода. Этим двум классам наследуют класс входных потоков *istream*, класс выходных потоков *ostream*. Производным от этих двух классов является класс *iostream*, определяющий двунаправленные потоки. Также этим двум классам наследуют:

- *ifstream* – входные файловые потоки;
- *ofstream* – выходные файловые потоки;
- *fstream* – двунаправленные файловые потоки;
- *istringstream* – входные строковые потоки;
- *ostringstream* – выходные строковые потоки;
- *stringstream* – двунаправленные строковые потоки.

Для использования в программе указанных классов с помощью директивы препроцессора *include* нужно включать следующие заголовочные файлы:

- *iostream* – содержит объявления классов входных и выходных потоков, а также стандартных потоков, таких как поток ввода *cin* (объект класса *istream*) и поток вывода *cout* (объект класса *ostream*), связанных с консолью;
- *fstream* – содержит объявления классов файловых потоков;
- *sstream* – содержит объявления классов строковых потоков.

Все классы и объекты библиотеки ввода-вывода входят в пространство имен *std*. Поэтому для их использования в программе нужно либо указывать уточненные имена (с префиксом *std::*), либо заранее размещать директиву *using*.

Включение в программу заголовочного файла *iostream* обеспечивает использование ряда стандартных потоков, создаваемых при запуске программы и обеспечивающих консольный ввод-вывод, в частности:

Имя потокового объекта	Класс	Описание
cin	istream	Стандартный поток ввода, по умолчанию ассоциируется с клавиатурой (стандартным буферизированным вводом)
cout	ostream	Стандартный поток вывода, по умолчанию ассоциируется с консолью (стандартным буферизированным выводом)
cerr	ostream	Стандартный поток ошибок, по умолчанию ассоциируется с консолью (стандартным небуферизированным выводом)
clog	ostream	Стандартный поток ошибок, по умолчанию ассоциируется с консолью (стандартным буферизированным выводом)

Стандартный поток *cerr* не буферизирован, это означает, что при выводе в него информация непосредственно попадает в консоль не

дожидаясь заполнения буфера или вывода символа перевода строки `'\n'`. Поток *clog* является его буферизированной версией.

Все упомянутые выше классы предназначены для ввода-вывода данных, состоящих из обычных 8-битных символов в кодировке ASCII типа *char*. Однако, в языке C++ доступна и обработка так называемых «широких», двухбайтных символов Unicode типа *wchar\_t*. Для этих целей у рассмотренных выше классов имеются аналоги, объявленные в тех же заголовочных файлах. Их названия начинаются с префикса «w», например: *wistream*, *wostream*, *wfstream* и т.д. Аналогично, доступны стандартные потоки для ввода-вывода «широких» символов: *wcin*, *wcout*, *wcerr*, *wclog*. Функционал классов и объектов для обработки стандартных и «широких» символов абсолютно одинаков. Далее он будет рассмотрен на примере классов и объектов, обрабатывающих однобайтные символы типа *char*.

### Средства потокового ввода

Для всех упомянутых выше классов входных потоков определена операция `>>` извлечения из потока. В предыдущих работах данная операция использовалась применительно к стандартному потоку *cin*. Операция перегружена для разных типов, что позволяет автоматически корректно обрабатывать символы из входного буфера сообразуясь с типом переменной. Пробелы, символы табуляции и перевода строки в любом количестве игнорируются и воспринимаются как разделители между значениями различных переменных. Значения вещественных переменных, как и целочисленных, по умолчанию вводятся в десятичной системе счисления. Вещественные переменные можно вводить в формате с фиксированной точкой (например, 1.234) или экспоненциальном (например, 12.34e-1) Значения целочисленных

переменных можно вводить в шестнадцатеричной или восьмеричной системах счисления (вводятся непосредственно сами значения, без префиксов констант «0x» и «0» соответственно). Для этого применяются манипуляторы *hex* и *oct* (определены в файле *iostream*). После их размещения все целочисленные переменные будут вводиться в соответствующей системе счисления. Возврат к десятичной системе осуществляется применением манипулятора *dec*. Например:

```
int a, b, c;  
std::cin >> std::hex >> a >> b; //ввод a и b в 16-чной СС  
std::cin >> std::dec >> c; //ввод c в 10-чной СС
```

Если с операцией извлечения из потока используется массив типа *char* (указатель), то осуществляется ввод строки. При этом вводится одна лексема, т.е. извлечение осуществляется до ближайшего пробела, символа табуляции или перевода строки (что встретится раньше). Вместо этих символов-разделителей в массив заносится нуль-символ '\0', ограничивающий текущую длину строки. Для ввода строк с пробелами и символами табуляции можно применять методы *get* и *getline* класса *istream* и его потомков, которые будут рассмотрены ниже.

Поскольку ввод значений переменных буферизирован, то вводимые символы попадают во входной буфер только после нажатия клавиши **Enter**. После этого начинается извлечение символов из буфера операцией *>>*. Если все символы соответствуют ожидаемым для данного типа текущей переменной, то считывание ее значения завершается штатно и после того, как встретился пробел, символ табуляции или перевода строки. В случае считывания символа, неподходящего под тип вводимой переменной, извлечение из входного буфера прекращается, значение текущей переменной формируется из тех корректных символов, что были извлечены ранее.

Для получения в программе информации о произошедшей ошибке ввода имеется несколько возможностей. В целом информация о наличии ошибки определяется, если использовать возвращаемую операцией извлечения >> ссылку как логическое выражение. В этом случае происходит автоматическое приведение типа и значение *false* сигнализирует о наличии ошибки при вводе:

```
double x;  
if(!(std::cin >> x)) std::cout << "Ошибка!";
```

Неверные символы при вводе переменной являются не единственными причинами возникновения ошибок в работе входных потоков. Для получения более подробной информации о характере проблемы в базовом классе *ios* имеется поле *state*, значение которого определяет состояние потока. Нулевое значение поля означает отсутствие ошибок или затруднений в работе потока. В противном случае в поле будет установлен в единицу один из битов, определенный константами: *eofbit* (по достижению конца файла), *badbit* (в случае ошибки чтения), *failbit* (в случае ошибки преобразования). Однако напрямую получить значение поля нельзя, для этого используются следующие методы:

```
bool eof(); // Возвращает true, если установлен eofbit  
bool fail(); // Возвращает true, если установлен failbit  
bool bad(); // Возвращает true, если установлен badbit  
int rdstate(); // Возвращает состояние потока
```

Например, проверить наличие ошибки преобразования после ввода переменных можно так:

```
if(std::cin.fail())  
    std::cout << "Ошибка преобразования";
```

Кроме операции `>>`, входные потоки имеют ряд методов неформатированного ввода. Например, для ввода отдельных символов (включая пробелы, символы табуляции и перевода строки) или строк с пробелами и символами табуляции может применяться метод `get()`. Метод перегружен и может применяться с разным количеством и типом параметров. Для ввода символов его можно применить следующим образом:

```
char c1, c2;  
c1 = std::cin.get();  
//или  
std::cin.get(c2);
```

При вводе строк задается буфер (массив символов), максимальное число считываемых символов (размерность буфера минус 1) и символ-разделитель (по умолчанию это символ перевода строки). Считывание строки продолжается до тех пор, пока в потоке не встретится символ-разделитель или символ конца файла. Затем в буфер дописывается нуль-символ `'\0'`. При этом сам символ-разделитель остается в потоке и может быть в дальнейшем обработан следующей операцией ввода. Например:

```
char s1[50], s2[50];  
std::cin.get(s1,49); //применен разделитель по умолчанию, т.е. '\n'  
//или  
std::cin.get(s2,49,';'); //разделитель – точка с запятой
```

Во многих случаях то, что встреченный символ-разделитель остается в потоке, является существенным неудобством. В таких случаях для ввода строки применяется другой метод – `getline()`. Возможный набор параметров у него совпадает с методом `get()`. Главным отличием является то, что встреченный в потоке

символ-разделитель извлекается и отбрасывается, т.е. не помещается в буфер. После ввода в буфер также дописывается нуль-символ.

Примеры:

```
char s3[50], s4[50];
std::cin.getline(s3,49);//разделитель - '\n'
//или
std::cin.getline(s4,49,'');//разделитель – точка с запятой
```

Если из потока нужно ввести определенное количество байт без какого-либо преобразования и дополнения буфера нуль-символом, то применяется метод *read()*. Данный метод обычно используется не для клавиатурного ввода, а для файлового, причем данные, как правило, предварительно записываются в файл с помощью метода *write()* класса *ostream* или его потомка. Это означает, что формат данных известен считывающей программе и они обрабатываются соответствующим образом как набор «сырых» байтов. В качестве параметров методу передаются буфер (массив символов) и его размерность.

Для иллюстрации рассмотрим ввод с помощью метода *read()* значения двухбайтовой беззнаковой переменной. Поскольку методу требуется передать буфер в виде указателя на *char*, представим такую переменную в виде массива из двух символов:

```
unsigned short h1;
std::cin.read((char*)&h1, sizeof(unsigned short));
```

Если при работе данного фрагмента ввести в консоли, к примеру, два нуля (00) и нажать **Enter**, то переменная *h1* получит значение 12336. Чтобы понять, почему два символа нуля превратились в такое значение вспомним, что ASCII символ '0' имеет код 30 в шестнадцатеричной системе счисления. Значит в каждом из двух байтов переменной будет записана эта величина. В двоичном виде шестнадцать битов



переменной будут иметь следующий вид: 0011 0000 0011 0000. Переведа значение в десятичную систему счисления получим 12336.

### **Средства потокового вывода**

Для всех упомянутых выше классов выходных потоков определена операция << помещения в поток. Операция перегружена для разных типов данных, что позволяет автоматически форматировать выводимое выражение в соответствии с его типом и помещать в буфер выходного потока.

По умолчанию значения вещественных выражений выводятся в поле шириной шесть символов. В зависимости от значения результат вывода может округляться, чтобы вместить число в поле шириной шесть символов. Если порядок числа меньше 6 или больше -5, то оно отображается в формате с фиксированной точкой, в противном случае – в экспоненциальном.

Целочисленные выражения по умолчанию отображаются как десятичные значения в поле достаточной ширины для их корректного отображения, в том числе и знака минус, если он имеется. Значения указателей (кроме указывающих на тип *char*) выводятся в шестнадцатеричной системе счисления.

Выражения типа *char* отображаются в виде печатаемого символа. Если требуется увидеть сам ASCII-код выражения, то его нужно приводить к целому типу, например к *int*.

Строки отображаются в поле шириной, равной длине строки (без учета нуль-символа). Операция помещения в поток определена так, что если ей передается указатель на тип *char*, то происходит вывод символов, начиная с первого (по адресу в указателе) до предшествующего нуль-символу. Для вывода самого адреса,

находящегося в указателе на тип *char*, его надо привести к указателю на тип *void*.

Если приведенные выше правила форматирования не устраивают, их можно изменять. Для этого в классах потокового вывода предусмотрено несколько возможностей. Далее будет кратко рассмотрена одна из них – использование манипуляторов.

Для смены системы счисления, в которой выводятся целочисленные выражения, используются уже упомянутые выше манипуляторы *hex*, *oct* и *dec*. Например, вывод константы в восьмеричной и десятичной системе счисления:

```
std::cout << std::oct << 512
          << ' ' << std::dec << 512 << std::endl;
```

В результате работы оператора в консоли появится:

```
1000 512
```

Расширенная настройка параметров форматирования задается с помощью флагов, представляющих собой отдельные биты в целочисленном поле *x\_flags* класса *ios*. Изменять флаги (напрямую это сделать невозможно) можно разными способами, в частности, для установки некоторых из них существуют отдельные манипуляторы. Примеры таких манипуляторов:

- *left* – устанавливает выравнивание по левому краю поля;
- *right* – устанавливает выравнивание по правому краю поля (установлено по умолчанию);
- *internal* – знак числа выводится с левого края, а само значение числа – с правого;
- *showbase* – отображает префиксы констант C++ при выводе восьмеричных и шестнадцатеричных значений («0» и «0x» соответственно);

- *uppercase* – вывод шестнадцатеричных значений и чисел в экспоненциальном формате производится с использованием прописных латинских букв;
- *nouppercase* – отключает предыдущий режим, т.е. происходит возврат к использованию строчных букв;
- *fixed* – вывод вещественных выражений производится в формате с фиксированной точкой, независимо от величины;
- *scientific* – вывод вещественных выражений производится в экспоненциальном формате, независимо от величины.

Для демонстрации изменений в формате вещественного значения при использовании двух последних манипуляторов рассмотрим вывод одной и той же константы в режиме по умолчанию, затем в формате с фиксированной точкой и в экспоненциальном виде:

```
std::cout << 12345.67 << ' ';
std::cout << std::fixed << 12345.67 << ' ';
std::cout << std::scientific << 12345.67 << '\n';
```

Результат такого вывода будет следующим:

```
12345.7 12345.670000 1.234567e+04
```

Рассмотренные выше манипуляторы не имеют параметров. Однако, кроме них существуют еще и параметризованные манипуляторы, объявления которых содержатся в заголовочном файле *iomanip*. Некоторые из этих манипуляторов:

- *setprecision()* – принимает в качестве аргумента целочисленное значение, задающее точность отображения вещественных величин: размерность дробной части для формата с фиксированной точкой и общее количество значащих цифр в экспоненциальном формате;

- *setw()* – устанавливает ширину поля в символах, равную целочисленному значению в скобках. Данный манипулятор действует однократно, только лишь для величины, выводимой следом за ним.

Для классов выходных потоков также определены дополнительные методы, расширяющие возможности вывода. С помощью метода *put()* в поток выводится отдельный символ. Отличие от возможностей операции помещения в поток << состоит в том, что аргумент метода необязательно может представлять собой выражение типа *char*, достаточно того, чтобы оно могло быть приведено к этому типу. В то время как операция помещения в поток выведет в поток символ только при явной принадлежности выражения к типу *char*. Например:

```
std::cout << 'a';  
std::cout.put(97);  
std::cout.put(97.7);  
std::cout << ' ' << 97;
```

В результате в консоль будет выведено:

```
aaa 97
```

Видно, что если операции помещения в поток передается символьная константа 'a', то в консоль выводится именно символ. Но если операции передать эквивалентное целое число (ASCII код латинской прописной буквы «a» – 97), то оно обрабатывается как целочисленное выражение. В то время как метод *put()* такое выражение (второй оператор) или вещественное (третий оператор) приводит к типу *char* и выводит в консоль соответствующий этому коду символ.

Метод *write()* осуществляет вывод заданного количества символов из указанного в качестве параметра массива типа *char*. Метод, как правило, используется в файловых потоках и выводит тот набор байтов,

который в дальнейшем будет считываться из файла с помощью рассмотренного выше метода *read()*. Для иллюстрации применения метода сделаем действия, обратные тем, которые приводились выше в качестве иллюстрации работы метода *read()*. Вначале присвоим двухбайтовой беззнаковой переменной значение 12336, затем выведем ее с помощью метода *write()*, приведя к типу указателя на *char*, т.е. представив в виде массива символов:

```
unsigned short h2 = 12336;  
std::cout.write((char*)&h2, sizeof(unsigned short));
```

В результате работы метода в консоль будет выведено два нуля (суть обратного преобразования приводится выше при рассмотрении метода *read()*).

## Файловые потоки

Как было сказано выше, для работы с файлами можно использовать следующие классы файловых потоков: *ifstream* (входные потоки), *ofstream* (выходные потоки), *fstream* (двунаправленные потоки). Для их использования в программу с помощью директивы препроцессора *include* нужно включить заголовочный файл *fstream*.

Только лишь объявить в программе переменную одного из этих классов недостаточно, нужно ассоциировать созданный объект с конкретным файлом на диске. Для этого имеются две возможности. Первая – это указать строку с именем файла в качестве параметра конструктора. Например, так можно открыть текстовый файл из корневого каталога диска C::

```
std::ifstream ifs1("c:\\in1.txt");
```

Другим вариантом является создание объекта файлового потока конструктором по умолчанию (без параметров), а затем применение метода *open()*. Аналогичный пример:

```
std::ifstream ifs2;  
ifs2.open("c:\\in2.txt");
```

Если используется класс *ifstream*, то по умолчанию поток открывается в режиме чтения. Файл при этом должен существовать. При использовании класса *ofstream* поток открывается для записи, если файл не существует по указанному пути, то он создается, если существует, то его прежнее содержимое удаляется. Класс *fstream* позволяет создать поток для чтения и записи одновременно, при этом его прежнее содержимое не удаляется и возможно его считывание.

В случае, если входной файл по заданному пути отсутствует или его невозможно открыть по каким-то другим причинам, если выходной файл невозможно создать, открытие потока завершается неудачей. Объект, который пытался установить с файлом связь, невозможно использовать для ввода-вывода. Чтобы узнать, завершилась ли успешно операция открытия потока, применяется метод *is\_open()*, который определен для всех упомянутых выше классов. Метод возвращает значение *false*, если попытка открытия файла закончилась неудачей:

```
if (!ifs1.is_open())  
    std::cout << "Ошибка открытия файла!" << std::endl;
```

В случае успешного открытия потока его можно использовать для ввода и/или вывода, в соответствии с его типом. Для этого применяются любые из рассмотренных выше операций и/или методов ввода-вывода, определенных для классов *istream* и *ostream*, так как файловые потоки им наследуют. После того, как ввод или вывод осуществлен, потоки нужно закрывать. Автоматическое закрытие всех потоков происходит

при завершении программы. При этом буфер выходного потока предварительно сбрасывает в файл свое содержимое. Но потоки можно закрывать и явно, для этого используется метод *close()*, определенный во всех классах файловых потоков:

```
ifs2.close();
```

Сама переменная при этом отключается от файла и может быть использована в дальнейшем для открытия другого файла с помощью метода *open()*.

Рассмотренные выше способы открытия потоков предполагали, что для них устанавливаются некоторые режимы работы по умолчанию, зависящие от класса объекта. Однако при открытии потока при любом из способов можно дополнительно указывать второй параметр, определяющий режим открытия файла. Данный параметр имеет некоторые значения по умолчанию, поэтому в рассмотренных выше примерах его можно было не указывать. Однако применение данного параметра дает дополнительные возможности при работе с файлами.

Параметр, определяющий режим, представляет собой целочисленное значение, равное одной из предопределенных констант или их комбинации, полученной с помощью операции побитового ИЛИ (*|*). Данные константы определены в классе *ios*:

- *ios:: in* – открытие файла для чтения;
- *ios:: out* – открытие файла для записи;
- *ios:: ate* – перейти к концу файла после его открытия;
- *ios:: app* – добавление в конец файла;
- *ios:: trunc* – удалить содержимое файла, если он существует;
- *ios:: binary* – открытие в двоичном режиме.

Для входных потоков режим по умолчанию определяет константа *ios:: in*, для выходных – выражение *ios:: out | ios:: trunc*. По умолчанию

входные и выходные потоки открываются в текстовом режиме, но также возможно открытие и в двоичном режиме добавлением к значению режима с помощью побитового ИЛИ константы *ios::binary*. Например, открыть файл для записи в корневом каталоге диска *C:* в двоичном режиме можно следующим образом:

```
std::ofstream ofs;  
ofs.open("c:\\out.txt", std::ios::out | std::ios::trunc |  
        std::ios::binary);
```

Двоичный режим применяется, как правило, тогда, когда данные требуется непосредственно считывать или записывать без каких-либо преобразований. В основном для этого применяются рассмотренные выше методы *read()* входных потоков и *write()* выходных потоков. В примере, иллюстрирующем применение метода *write()*, беззнаковая целочисленная переменная размером два байта, имеющая десятичное значение 12336, выводится данным методом в консоль в виде двух символов '0'. Это значит, что при выводе в файл такая переменная будет занимать в нем именно два байта, всегда и независимо от ее конкретного значения. Если же для вывода переменной воспользоваться операцией помещения в поток <<, то в файле в ASCII кодировке ее отображение займет пять байт, по количеству символов, представляющих каждый значащий десятичный разряд. Т.е., вывод в файл «сырых» байт без преобразования позволит, во многих случаях, уменьшить его объем, хотя и затруднит при этом анализ его содержимого при просмотре в текстовом редакторе.

Еще одной особенностью двоичных потоков является обработка управляющего символа новой строки '\n' при вводе-выводе, которая напрямую зависит от используемой операционной системы. В частности, в Windows, при работе в текстовом редакторе с символами в



кодировке ASCII нажатие клавиши **Enter** приводит к генерации двух символов: возврат каретки, перевод строки. Когда при выводе в текстовый поток в него помещается управляющий символ новой строки ‘\n’, то он автоматически заменяется в файле на пару символов перевода строки и возврата каретки. При вводе происходит обратное преобразование, поэтому после считывания пары символов из файла непосредственно в поток для программной обработки помещается только один символ новой строки. Однако, если при выводе «сырых» байт какой-то из них будет иметь значение, равное коду символа новой строки, то в текстовом потоке произойдет описанная выше замена, что, безусловно, исказит выводимые данные. Аналогичная замена может произойти и при вводе данных.

Таким образом, если осуществляется файловый ввод данных, заранее подготовленных в текстовом редакторе или вывод результатов работы программы, с целью дальнейшего их просмотра или печати, то нужно применять текстовые потоки (по умолчанию) и операции извлечения из потока, помещения в поток и те методы, которые обеспечивают необходимые преобразования. Этот вариант файлового ввода-вывода используется в большинстве случаев. Если же данные нужно вводить или выводить без каких-либо преобразований, то требуется применять двоичные потоки и методы *read()* и *write()*.

Для иллюстрации работы с текстовыми входными и выходными потоками рассмотрим решение следующей задачи.

*Составить программу, считывающую из файла in.txt значение целочисленной переменной N, элементы матрицы  $A(N \times N)$ , сохраняющую их в динамическом массиве и записывающую в файл out.txt транспонированную матрицу.*

```
#include <iostream>
```

```

#include <iomanip>
#include <fstream>
#include <locale>
using namespace std;
int main() {
    setlocale(LC_CTYPE, "");
    ifstream ifs("in.txt");
    //Проверка открытия входного файла, останов в случае ошибки:
    if (!ifs.is_open()) {
        cout << "Ошибка открытия входного файла" << endl;
        return 1;
    }
    unsigned n,i,j;
    //Считывание размерности матрицы, останов в случае ошибки:
    if (!(ifs >> n)) {
        cout << "Ошибка чтения данных из файла" << endl;
        return 1;
    }
    //Создание динамического массива и ввод элементов из файла:
    int** a=new int *[n];
    for (i = 0; i < n; i++) {
        a[i] = new int[n];
        for (j = 0; j < n; j++)
            //При ошибке считывания элементов
            //матрицы из файла – останов:
            if (!(ifs >> a[i][j])) {
                cout << "Ошибка чтения данных из\
                    файла" << endl;
                return 1;
            }
    }
}

```

```

    }
    //Закрытие входного файла:
    ifs.close();
    ofstream ofs;
    ofs.open("out.txt");
    //Проверка создания выходного файла, останов в случае ошибки:
    if (!ofs.is_open()) {
        cout << "Ошибка создания выходного файла" <<
endl;

        return 1;
    }
    //Вывод транспонированной матрицы в файл:
    ofs << "Транспонированная матрица:" << endl;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            ofs << setw(6) << a[j][i];
        ofs << endl;
    }
    //Освобождение памяти, занимаемой динамическим массивом:
    delete[]a;
    //Закрытие выходного файла:
    ofs.close();
    cout << "Работа программы завершена" << endl;
    return 0;
}

```

Если разместить в папке проекта, содержащей файл с исходным кодом, текстовый файл *in.txt* со следующим содержимым:

```

5
1 2 3 4 5
6 7 8 9 10

```

11 12 13 14 15

16 17 18 19 20

21 22 23 24 25

то после запуска в этой же папке будет создан файл *out.txt*:

Транспонированная матрица:

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

## СОДЕРЖАНИЕ РАБОТЫ

1. Выбрать алгоритм, составить его блок-схему и программу для решения варианта задания (см. ниже). Программа должна запрашивать и осуществлять ввод с клавиатуры имени файла с исходными данными, открывать файл для чтения, считывать из него размерность исходной квадратной матрицы (должна находиться в первой строке) и ее элементы (из последующих строк), после чего производить их требуемую обработку. Для хранения данных использовать динамические массивы.
2. Программа должна выводить в консоль и в файл исходную матрицу и результирующий массив. Имя выходного файла должно запрашиваться и вводиться с клавиатуры.

## ВАРИАНТЫ ЗАДАНИЯ

1. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов соответствующей строки верхней треугольной матрицы.

2. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен произведению элементов соответствующего столбца нижней треугольной матрицы.
3. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен наибольшему элементу соответствующей строки матрицы.
4. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если произведение элементов соответствующего столбца больше нуля и  $-1$  в противном случае.
5. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен наименьшему из элементов соответствующего столбца матрицы.
6. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если количество положительных элементов в соответствующей строке больше количества отрицательных и  $-1$  в противном случае.
7. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен среднему арифметическому наибольшего и наименьшего из элементов соответствующего столбца матрицы.
8. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если в соответствующей строке элемент главной диагонали больше элемента побочной и  $-1$  в противном случае.
9. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если элементы упорядочены по возрастанию или по убыванию и  $-1$  в противном случае.

10. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен первому отрицательному элементу соответствующей строки матрицы или нулю, если все элементы строки положительны.
11. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если все элементы соответствующей строки положительны и  $-1$  в противном случае.
12. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен соответствующему элементу столбца с наибольшей среди других столбцов суммой положительных элементов.
13. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если наименьший элемент соответствующей строки положителен и  $-1$  в противном случае.
14. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов соответствующей строки, если они все либо положительны либо отрицательны, и нулю в противном случае.
15. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен среднему арифметическому элементов строки и столбца, на пересечении которых находится соответствующий элемент побочной диагонали.
16. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен количеству вхождений в соответствующую строку наибольшего из элементов матрицы.
17. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов того столбца, в

котором находится первый положительный элемент соответствующей строки, и нулю, если все элементы строки неположительны.

18. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если в соответствующем столбце есть возрастающая подпоследовательность из трех элементов и нулю в противном случае.
19. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если наименьший из элементов соответствующей строки совпадает с наименьшим элементом матрицы и  $-1$  в противном случае.
20. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен элементам того столбца, в котором находятся наибольший и наименьший по модулю элементы матрицы, или элементам побочной диагонали, если они находятся в разных столбцах.
21. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов соответствующей строки, предшествующих первому в ней отрицательному элементу.
22. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если сумма модулей элементов соответствующего столбца больше наибольшего по модулю элемента матрицы и  $-1$  в противном случае.
23. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен произведению элементов

соответствующего столбца, расположенных за первым в нем отрицательным элементом.

24. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если количество четных элементов в соответствующей строке больше количества нечетных и  $-1$  в противном случае.

25. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если наибольший по модулю элемент соответствующей строки совпадает с наименьшим по модулю элементом побочной диагонали и  $-1$  в противном случае.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Каковы в языке C++ принципы размещения в памяти многомерных массивов? Как производится их описание?
2. Как производится обращение к элементам многомерного массива?
3. Какими способами можно произвести заполнение многомерного массива элементами?
4. Что такое динамическая переменная?
5. Какие средства для распределения динамической памяти вы знаете?
6. Каким образом в языке C++ создаются динамические массивы?
7. Каким образом в языке C++ освобождается память, занимаемая динамическим массивом?
8. Какие классы потокового ввода-вывода в языке C++ вы знаете?
9. В каких заголовочных файлах определены классы потокового ввода-вывода?
10. Какие потоки создаются при запуске программы?



11. Какие операции определены для входных и выходных потоков?
12. Какие методы классов входных и выходных потоков вам известны?
13. Как открываются файловые потоки?
14. Как закрываются файловые потоки?
15. Как меняется режим открытия файлового потока?
16. Какие режимы открытия файловых потоков вам известны?

## Лабораторная работа №7

# ПРИМЕНЕНИЕ ИТЕРАТИВНЫХ И РЕКУРСИВНЫХ ФУНКЦИЙ

**Цель работы:** ознакомиться с организацией передачи параметров в функции по ссылке; получить навыки описания рекурсивных функций.

## КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

### Описание функций

Определение функции состоит из ее заголовка и составного оператора тела функции (блока). В заголовке указывается тип функции, ее имя и в скобках список формальных параметров (если они есть). Параметры любых типов (кроме массивов и указателей) в обычном случае передаются в функции по значению. Подразумевается, что функция может возвращать результат своей работы объявленного при ее описании типа через свое имя (если она не типа *void*). Функция может возвращать значение любого типа, кроме массивов и функций. Но функция также может возвращать указатель на объект любого типа, в том числе и на массив или функцию.

К месту своего первого вызова в программе функция уже должна быть определена. Если по каким-то причинам разместить определение перед вызовом не получается, сначала добавляется прототип функции, содержащий ее заголовок и заканчивающийся точкой с запятой. Причем в прототипе имена формальных параметров могут не указываться, а просто перечисляться названия типов.

В некоторых случаях, например, при необходимости возврата нескольких значений, применяется передача параметров по ссылке. Для этого имеются две возможности:

### **1. Ссылочные параметры.**

*Пример:* функция инкремента целого числа:

```
void inc(int &x)
{
    x++;
}
.....
```

*// Вызов функции:*

```
int x=1;
inc(x);
```

### **2. Указатели.**

*Пример:* функция инкремента целого числа:

```
void inc(int *x)
{
    (*x)++;
}
.....
```

*// Вызов функции:*

```
int x=1;
inc(&x);
```

Поскольку имя массива является константным указателем, то в качестве параметра он *всегда* передается в функцию по ссылке. В случае одномерного массива в функцию фактически передается адрес первого элемента. Для статического одномерного массива возможны следующие варианты описания в качестве параметра функции:

```
void func1(int arr1[], int n);
```

```
// Или
```

```
void func2(int *arr2, int n);
```

По сути, первый вариант описания параметра-массива сводится ко второму, так как в функцию передается указатель на базовый тип. Поэтому размерность статического массива в квадратных скобках не указывается, если ее указать, она будет проигнорирована. Для того чтобы в функции при обработке массива оперировать его размерностью, нужно либо использовать дополнительный параметр, передаваемый по значению (как показано в данных прототипах), либо иметь общую для программы константу, доступную во всех функциях и применять ее в качестве размера.

Если массив динамический, то в функцию всегда передается указатель на базовый тип и размер массива в качестве отдельного параметра, как указано в прототипе функции *func2*.

Двумерный массив представляет собой массив, элементами которого являются одномерные массивы (строки). При вызове функции, имеющей в качестве параметра статический двумерный массив, ей будет передаваться указатель на первую строку матрицы. В связи с этим также возможны разные варианты указания двумерного массива в качестве параметра: традиционный, с использованием двух пар квадратных скобок и указанием во второй количества столбцов или в виде указателя на одномерный массив, размер которого совпадает со второй размерностью матрицы. Ниже показаны примеры передачи в функцию матриц со второй размерностью равной пяти. Количество строк в матрице также передается в виде отдельного параметра:

```
void func3(double matr1[][5], int n);
```

```
// Или
```

```
void func4(double (*matr2)[5], int n);
```

Двумерные динамические массивы передаются в функции через двойные указатели (указатели на указатели) на базовый тип. Отдельно передаются размерности массива, например:

```
void func5(double **matr3, int n, int m);
```

Если среди параметров, передаваемых в функцию по ссылке, имеются такие, которые не должны изменяться в процессе ее работы, перед ними при описании указывается модификатор *const* (за исключением случаев, когда параметр является двойным указателем).

Как было сказано выше, функция в качестве результата может возвращать указатель на функцию. Но также верным является то, что такой указатель может быть параметром функции. Общий синтаксис описания указателя на функцию следующий:

*<тип ф-ии> (\*<имя указателя>)(<список типов парам-в ф-ии>)*

Например, если имеется такое определение функции:

```
int sum(int a, int b)
{
    return a+b;
}
```

то указатель, ссылающийся на нее, может определяться так:

```
int (*pf)(int, int)=&sum;
```

Пример:

*Из элементов целочисленной матрицы  $A(N \times N)$  определить массив  $X$  из  $N$  элементов, равных элементам побочной диагонали матрицы  $A$ . Элементы матрицы по выбору пользователя либо вводятся с клавиатуры, либо генерируются случайным образом. Значение  $N$  вводится с клавиатуры. Для хранения данных использовать динамические массивы. Исходную матрицу и результирующий массив вывести в консоль.*

```

#include <iostream>
#include <iomanip>
#include <locale>
//Содержит прототипы функций работы с ГСЧ:
#include <cstdlib>
//Содержит прототипы функций работы с временем:
#include <ctime>
using namespace std;
//Прототип функции создания матрицы и ее ввода с клавиатуры:
int** create_matr_cons(int);
//Прототип функции создания матрицы и ее заполнения с помощью ГСЧ:
int** create_matr_rand(int);
//Прототип функции формирования массива X:
int* create_mas_x(int**, int);
//Прототип функции вывода матрицы и массива X:
void output(int** (*pfunc)(int), int);
int main() {
    setlocale(LC_CTYPE, "");
    int n, choice;
    int** (*pfunc)(int);
    cout << "Введите размерность матрицы:";
    cin >> n;
    do {
        cout << "Выбор способа создания матрицы:\n" <<
            "1 - клавиатура, 2 - ГСЧ\n";
        cin >> choice;
    } while (choice != 1 && choice != 2);
    //Присваивание указателю адреса одной из функций ввода матрицы:
    if (choice == 1)
        pfunc = &create_matr_cons;

```

```

        else
            pfunc = &create_matr_rand;
        output(pfunc, n);
    }

int** create_matr_cons(int n)
{
    int** a = new int* [n];
    cout << "Введите элементы матрицы:\n";
    for (int i = 0; i < n; i++) {
        a[i] = new int[n];
        for (int j = 0; j < n; j++)
            cin >> a[i][j];
    }
    return a;
}

int** create_matr_rand(int n)
{
    int** a = new int* [n];
    //Инициализация ГСЧ. time() возвращает текущее время в секундах:
    srand((unsigned)time(nullptr));
    for (int i = 0; i < n; i++) {
        a[i] = new int[n];
        for (int j = 0; j < n; j++)
            //Генерация псевдослучайных чисел от -19 до 30:
            a[i][j] = 30-rand()%50;
    }
    return a;
}

```

```

int* create_mas_x(int** a, int n)
{
    int* x = new int[n];
    for (int i = 0, j = n - 1; i < n; i++, j--)
        x[i] = a[i][j];
    return x;
}

void output(int** (*pfunc)(int), int n)
{
    int** a = pfunc(n);
    int* x = create_mas_x(a, n);
    cout << "Исходная матрица:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << setw(6) << a[i][j];
        cout << endl;
    }
    cout << "Массив X:\n";
    for (int i = 0; i < n; i++)
        cout << setw(6) << x[i];
    delete[]a;
    delete[]x;
}

```

## Рекурсивные функции

Функция называется *рекурсивной*, если она вызывает саму себя (прямая рекурсия), или вызывает другую функцию, которая в свою очередь вызывает первую (косвенная рекурсия).



Глубина рекурсии должна быть конечной. При выполнении очередного рекурсивного вызова система создает в стеке новые экземпляры всех автоматических переменных функции и ее параметров. Поэтому при слишком большой глубине рекурсии возможно переполнение стека и аварийное завершение работы программы.

Также следует аккуратно обращаться в рекурсивных функциях с глобальными переменными, так как их изменение отразится во всех последующих вызовах.

Структура простой функции с одним рекурсивным вызовом следующая. Имеется некоторое условие прекращения рекурсивных вызовов. Пока условие не сработало, происходят рекурсивные вызовы, в которых хотя бы один из параметров имеет значение, отличное от предыдущего вызова. Обычно стратегия рекурсивного решения задачи такова, что оно выражается в терминах решения более простого варианта задачи и существует решение самого простого варианта. Когда срабатывает условие прекращения рекурсивных вызовов, то реализуется самый простой вариант решения, после чего новые вызовы прекращаются и каждый из уже произведенных, начиная с последнего, должен доработать до конца функции и вернуться в предыдущий вызов. Таким образом, постепенно произойдет возврат в ту функцию, из которой первоначально осуществлен вызов рекурсивной функции.

В теле рекурсивной функции операторы могут располагаться и до точки рекурсивного вызова, и после. В первом случае операторы будут выполняться на так называемом *рекурсивном спуске*, во втором – на *рекурсивном подъеме*. Операторы спуска выполняются до того, как сработало условие прекращения рекурсивных вызовов, а операторы подъема – после.

Пример:

*Составить программу, содержащую рекурсивную функцию вычисления факториала. Программа должна вводить с клавиатуры аргумент и выводить результат в консоль.*

```
#include <iostream>
#include <locale>
using namespace std;
int factorial(unsigned n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main() {
    unsigned n;
    setlocale(LC_CTYPE, "");
    cout << "Введите n: ";
    cin >> n;
    cout <<n<< "! = " << factorial(n) << endl;
}
```

Для того чтобы лучше понять принцип работы данной программы, воспользуемся средствами отладки, такими как пошаговое выполнение, просмотр значений переменных и параметров, просмотр стека вызовов функций. Создадим новый проект, в который введем составленную программу. Далее с помощью клавиши **F11** (шаг с заходом в функции) начнем пошаговое выполнение программы. Выполнив инструкцию ввода, введем в консоли значение переменной *n* равное 3. При выполнении последнего оператора в теле функции *main()* происходит вызов рекурсивной функции *factorial()*, которой в качестве

фактического параметра передается ранее введенное значение переменной  $n$  (3). После передачи функции управления, в ее теле также начинают пошагово выполняться операторы. В частности, проверяется условие прекращения рекурсивных вызовов. Поскольку текущее значение параметра функции не равно 1 или 0, в условном операторе срабатывает ветвь *else*, где находится оператор возврата *return* с выражением, содержащим рекурсивный вызов функции *factorial()*. Момент перед выполнением данного оператора показан на рис. 7.1.

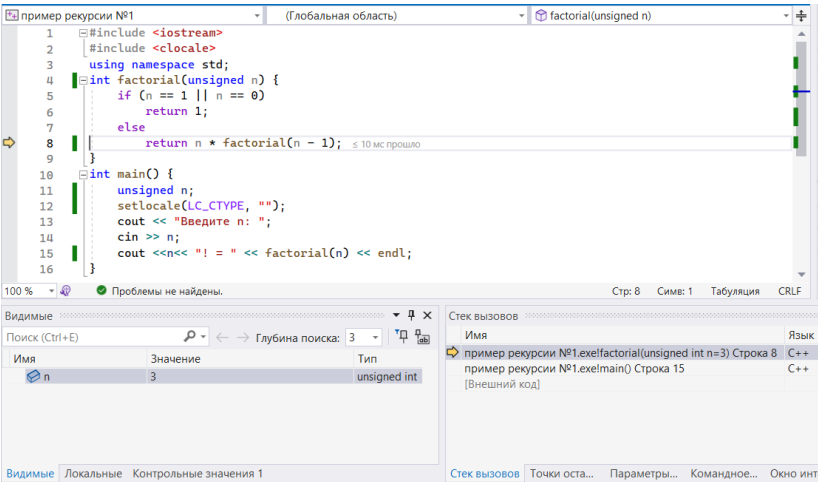


Рис. 7.1 Первый вызов рекурсивной функции при пошаговом исполнении программы

В информационном окне, на вкладке *Видимые* отображается текущее значение параметра  $n$ , равное 3. Справа на вкладке *Стек вызовов* отображается список произведенных вызовов функций. Видно, что выполнение программы началось с функции *main()*, затем был произведен вызов функции *factorial()*, который является на данный момент текущим. При нажатии клавиши **F11** будет осуществлен второй рекурсивный вызов функции (в составе выражения оператора *return*), но

уже с параметром  $n$  равным 2. В новом вызове при проверке условия управление также будет передано ветви *else* условного оператора и будет осуществлен еще один рекурсивный вызов функции *factorial()* с параметром  $n$  равным 1. При выполнении третьего рекурсивного вызова логическое выражение в условном операторе будет иметь значение *true*, в результате чего рекурсивные вызовы прекращаются и выполняется оператор *return*, возвращающий значение 1 (факториал единицы). На рис. 7.2 показан момент перед выполнением данного оператора.

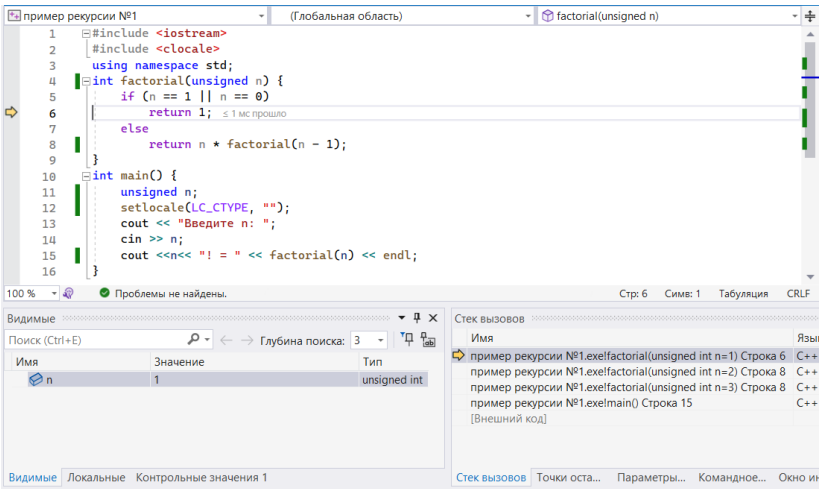


Рис. 7.2 Последний вызов рекурсивной функции при пошаговом исполнении программы

На вкладке *Видимые* отображается текущее значение параметра  $n$ , равное 1. На вкладке *Стек вызовов* видны все три рекурсивных вызова функции, произведенные к текущему моменту, и параметры, с которыми они осуществлялись. После очередного нажатия клавиши **F11** будет осуществлен переход к закрывающей фигурной скобке тела функции. Это означает, что текущий вызов отработал до конца и при новом нажатии клавиши **F11** произойдет возврат к предыдущему

вызову (с параметром  $n$  равным 2). Изменения отразятся в содержимом вкладок *Видимые* (дополнительно появится значение, возвращенное предыдущим вызовом) и *Стек вызовов* (из списка удалится последний вызов). Аналогичным образом завершим выполнение второго рекурсивного вызова и управление вернемся к первому вызову. На рис. 7.3 показан момент возврата в первый вызов.

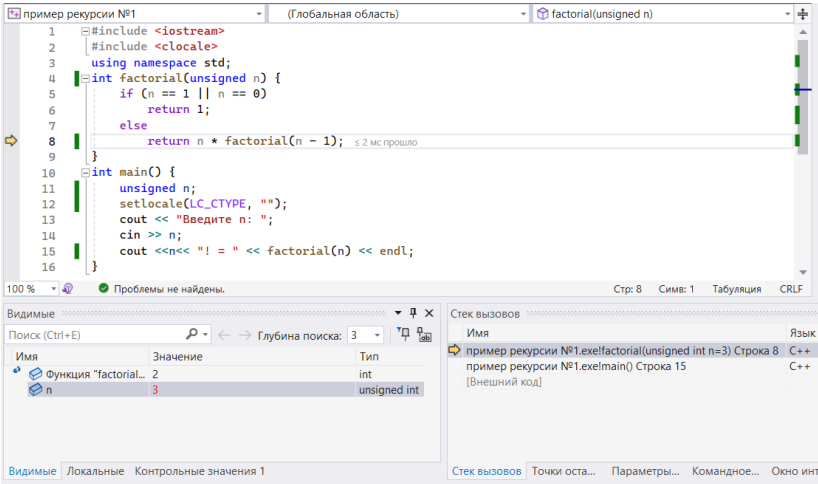


Рис. 7.3 Возврат к первому вызову рекурсивной функции при пошаговом исполнении программы

Во вкладке *Видимые* отображается значение 2, возвращенное предыдущим вызовом и текущее значение параметра  $n$  (3). Во вкладке *Стек вызовов* виден только первый вызов рекурсивной функции. Остальные рекурсивные вызовы к данному моменту уже завершены.

При последующем нажатии клавиши **F11** произойдет возврат в функцию `main()`. Момент возврата показан на рис. 7.4.

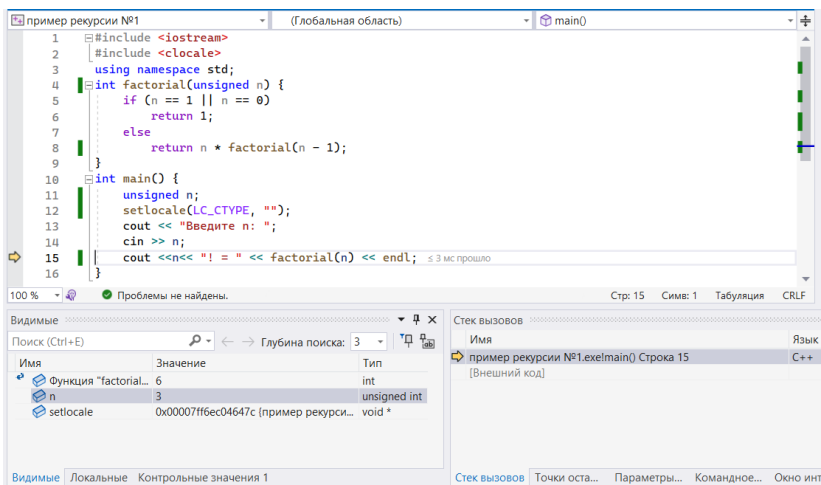


Рис. 7.4 Возврат в функцию *main()* при пошаговом исполнении программы

Вкладка *Видимые* отображает значение локальной переменной *n* (3), а также значение, возвращенное по итогу функцией *factorial()* (6). При последующих нажатиях клавиши **F11** завершится выполнение инструкции вывода, в консоли появится результат работы программы, после чего завершится выполнение тела функции *main()* и программы в целом.

## СОДЕРЖАНИЕ РАБОТЫ

1. Выбрать алгоритм, составить его блок-схему и программу для решения варианта задания (см. ниже). Программа должна по выбору пользователя осуществлять ввод исходной квадратной матрицы с клавиатуры или из файла, производить ее обработку согласно задания и осуществлять вывод исходных данных и результатов в консоль и в файл. Для этого программа должна содержать следующие функции:

- создания матрицы (динамического массива) и ввода ее элементов с клавиатуры,
  - создания матрицы (динамического массива) и ввода ее элементов из файла,
  - формирования на основе элементов матрицы динамического массива  $X$ ,
  - формирования величины  $Y$  (рекурсивная),
  - вывода исходной матрицы, массива  $X$ , величины  $Y$  в консоль и в файл,
  - функцию *main()*.
2. Функциям создания матрицы необходимо в качестве параметра передавать ее размерность. Функция для ввода из файла должна в процессе своей работы запрашивать и осуществлять ввод его имени с клавиатуры, открывать файл для чтения и считывать из него элементы квадратной матрицы заданной размерности. Обе функции должны возвращать созданную матрицу в качестве своего значения (как двойной указатель на базовый тип).
  3. Функция формирования массива  $X$  должна принимать в качестве параметров исходную матрицу (как двойной указатель на базовый тип) и ее размерность, а возвращать в качестве своего значения созданный массив (как указатель на базовый тип).
  4. Рекурсивная функция должна принимать в качестве параметров в зависимости от варианта задания или матрицу  $A$  или массив  $X$ , размерность матрицы/массива, а также необходимые дополнительные параметры, позволяющие организовать рекурсивное вычисление величины  $Y$  (продумать самостоятельно, в зависимости от варианта задания). Функция

должна возвращать в качестве своего значения вычисленную величину  $Y$ .

5. Функция вывода исходной матрицы, результирующего массива  $X$  и величины  $Y$  должна принимать в качестве параметра указатель на функцию ввода и размерность матрицы. В процессе своей работы функция должна вызывать через указатель одну из функций создания и ввода исходной матрицы, функцию формирования массива  $X$ , рекурсивную функцию вычисления величины  $Y$ . Также функция должна запрашивать имя выходного файла, открывать его для записи и выводить в консоль и в файл исходную матрицу  $A$ , результирующий массив  $X$ , вычисленную величину  $Y$ .
6. Функция *main()* должна осуществлять ввод с клавиатуры размерности матрицы, запрашивать способ ввода матрицы, в зависимости от которого осуществлять вызов функции вывода с указателем на одну из функций ввода в качестве параметра.
7. В программе не должно быть глобальных переменных.

### **ВАРИАНТЫ ЗАДАНИЯ**

1. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов соответствующей строки верхней треугольной матрицы. Определить величину  $Y$ , как наибольший из элементов массива  $X$ .
2. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен произведению элементов соответствующего столбца нижней треугольной матрицы. Определить величину  $Y$ , как сумму положительных элементов массива  $X$ .



3. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен наибольшему элементу соответствующей строки матрицы. Определить величину  $Y$ , как наименьший из положительных элементов массива  $X$ .
4. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если произведение элементов соответствующего столбца больше нуля и  $-1$  в противном случае. Определить величину  $Y$ , как количество повторений 1 среди элементов массива  $X$ .
5. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен наименьшему из элементов соответствующего столбца матрицы. Определить величину  $Y$ , как количество нечетных элементов, расположенных перед наибольшим из элементов массива  $X$ .
6. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если количество положительных элементов в соответствующей строке больше количества отрицательных и  $-1$  в противном случае. Определить величину  $Y$ , как количество четных элементов в первой строке матрицы  $A$ .
7. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен среднему арифметическому наибольшего и наименьшего из элементов соответствующего столбца матрицы. Определить величину  $Y$ , как сумму элементов, расположенных перед наименьшим элементом массива  $X$ .

8. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если в соответствующей строке элемент главной диагонали больше элемента побочной и  $-1$  в противном случае. Определить величину  $Y$ , как количество нечетных элементов в первом столбце матрицы  $A$ .
9. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если элементы упорядочены по возрастанию или по убыванию и  $-1$  в противном случае. Определить величину  $Y$ , как среднее арифметическое наибольшего и наименьшего элемента главной диагонали матрицы  $A$ .
10. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен первому отрицательному элементу соответствующей строки матрицы или нулю, если все элементы строки положительны. Определить величину  $Y$ , как индекс первого отрицательного элемента массива  $X$ .
11. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если все элементы соответствующей строки положительны и  $-1$  в противном случае. Определить величину  $Y$ , как 1, если элементы первой строки матрицы образуют арифметическую прогрессию и 0 в противном случае.
12. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен соответствующему элементу столбца с наибольшей среди других столбцов суммой положительных элементов. Определить величину  $Y$ , как 1, если элементы

массива  $X$  образуют последовательность Фибоначчи ( $f_1 = f_2 = 1$ ,  $f_i = f_{i-1} + f_{i-2}$  для  $i > 2$ ) и 0 в противном случае.

13. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если наименьший элемент соответствующей строки положителен и  $-1$  в противном случае. Определить величину  $Y$ , как наибольший из индексов элементов массива  $X$ , равных 1.
14. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов соответствующей строки, если они все либо положительны либо отрицательны, и нулю в противном случае. Определить величину  $Y$ , как сумму элементов массива  $X$ , расположенных после первого нулевого элемента.
15. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен среднему арифметическому элементов строки и столбца, на пересечении которых находится соответствующий элемент побочной диагонали. Определить величину  $Y$ , как произведение четных элементов, расположенных после наименьшего элемента массива  $X$ .
16. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен количеству вхождений в соответствующую строку наибольшего из элементов матрицы. Определить величину  $Y$ , как количество нулевых элементов массива  $X$ .
17. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов того столбца, в котором находится первый положительный элемент

соответствующей строки, и нулю, если все элементы строки неположительны. Определить величину  $Y$ , как количество отрицательных элементов, расположенных перед наибольшим элементом массива  $X$ .

18. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если в соответствующем столбце есть возрастающая подпоследовательность из трех элементов и нулю в противном случае. Определить величину  $Y$ , как произведение нечетных элементов, расположенных перед первым встретившимся четным элементом массива  $X$ .
19. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если наименьший из элементов соответствующей строки совпадает с наименьшим элементом матрицы и  $-1$  в противном случае. Определить величину  $Y$ , как сумму четных элементов первой строки матрицы, расположенных после первого нечетного элемента
20. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен элементам того столбца, в котором находятся наибольший и наименьший по модулю элементы матрицы, или элементам побочной диагонали, если они находятся в разных столбцах. Определить величину  $Y$ , как произведение отрицательных элементов массива  $X$ , расположенных после первого положительного элемента.
21. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен сумме элементов соответствующей строки, предшествующих первому в ней отрицательному

- элементу. Определить величину  $Y$ , как количество повторений наименьшего элемента в первой строке матрицы.
22. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если сумма модулей элементов соответствующего столбца больше наибольшего по модулю элемента матрицы и  $-1$  в противном случае. Определить величину  $Y$ , как сумму положительных элементов первой строки матрицы, расположенных после первого нулевого элемента.
23. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен произведению элементов соответствующего столбца, расположенных за первым в нем отрицательным элементом. Определить величину  $Y$ , как количество отрицательных элементов первой строки матрицы, имеющих нечетные номера столбцов.
24. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если количество четных элементов в соответствующей строке больше количества нечетных и  $-1$  в противном случае. Определить величину  $Y$ , как произведение положительных элементов, расположенных после наибольшего элемента первой строки матрицы.
25. Дана матрица  $A(N \times N)$ . Определить массив  $X$  из  $N$  элементов, каждый из которых равен 1, если наибольший по модулю элемент соответствующей строки совпадает с наименьшим по модулю элементом побочной диагонали и  $-1$  в противном случае. Определить величину  $Y$ , как сумму элементов первой

строки матрицы, расположенных между наибольшим и наименьшим элементом.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Какие функции осуществляют открытие и закрытие файла?
2. Какие функции предназначены для форматированного ввода-вывода данных?
3. В чем заключается различие в принципах передачи в функцию параметров по значению и по ссылке?
4. Какие вы знаете способы передачи параметров по ссылке в языке C/C++?
5. Каким образом передаются в функции массивы?
6. Возможен ли возврат функцией таких типов данных, как структуры и объединения?
7. Возможен ли возврат функцией массива?
8. Назовите преимущества и недостатки рекурсивных функций по сравнению с итеративными.
9. В каком случае задача может иметь рекурсивное решение?
10. Каков механизм вызова рекурсивной функции?
11. Какие условия должны выполняться при описании рекурсивных функций?
12. Как описываются функции с косвенной рекурсией?

## Лабораторная работа №8

### ПОБИТОВЫЕ ОПЕРАЦИИ ЯЗЫКА C++

**Цель работы:** получение навыков использования побитовых операций при работе с целочисленными объектами.

#### КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Побитовые операции производятся над всеми разрядами (битами) двоичного представления операндов. Побитовые операции применимы только к тем переменным и выражениям, которые имеют целочисленное внутреннее представление (т.е. к целым и символьным). Если операции производятся над операндами знакового типа, то необходимо учитывать, что отрицательные значения представляются в дополнительном коде и старший (знаковый) бит также участвует во всех операциях. В языке C++ имеются следующие побитовые операции (в порядке убывания приоритета):

1. **Побитовое отрицание** ( $\sim$ ) – производит побитовую инверсию двоичного представления операнда.

*Пример:* побитовое отрицание числа 250 ( $\sim 250$ ):

Исходное представление числа 250 (беззнаковое, однобайтное):

1	1	1	1	1	0	1	0
---	---	---	---	---	---	---	---

Результат:

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

Или  $5_{10}$ .

2. **Сдвиг** на определенное число разрядов: влево ( $\ll$ ) и вправо ( $\gg$ ).

*Пример:*

а) сдвиг числа 48 на два разряда влево ( $48 \ll 2$ ):

Исходное представление числа 48 (беззнаковое, однобайтное):

0	0	1	1	0	0	0	0
---	---	---	---	---	---	---	---

Результат:

1	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Или  $192_{10}$ . Как видно сдвиг числа влево на  $n$  разрядов приводит к его умножению на  $2^n$ ;

б) сдвиг числа 48 на два разряда вправо ( $48 \gg 2$ ):

Исходное представление числа 48 (беззнаковое, однобайтное):

0	0	1	1	0	0	0	0
---	---	---	---	---	---	---	---

Результат:

0	0	0	0	1	1	0	0
---	---	---	---	---	---	---	---

Или  $12_{10}$ . Как видно сдвиг числа вправо на  $n$  разрядов приводит к его делению на  $2^n$ .

Сдвиг не является циклической операцией и если происходит выход какого-либо бита за пределы разрядной сетки, то его значение теряется. Освобождающиеся при сдвиге разряды заполняются нулями. Если производится сдвиг вправо операнда знакового типа, то дополнение слева производится содержимым старшего бита.

3. **Побитовое И (&)** – производится над двумя операндами и если оба соответствующих бита равны единице, то результат – единица, в противном случае – ноль.

*Пример: 202 & 83:*

202:

1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---

&



83:

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

Результат:

0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---

Или 66<sub>10</sub>.

4. **Побитовое исключающее ИЛИ (^)** – производится над двумя операндами и если оба соответствующих бита равны единице или нулю, то результат – ноль, в противном случае – единица.

*Пример:* 202 ^ 83:

202:

1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---

^

83:

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

Результат:

1	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---

Или 153<sub>10</sub>.

5. **Побитовое ИЛИ (|)** – производится над двумя операндами и если оба соответствующих бита равны нулю, то результат – нуль, в противном случае – единица.

*Пример:* 202 | 83:

202:

1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---

|

83:

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

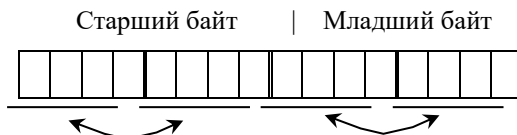
Результат:

1	1	0	1	1	0	1	1
---	---	---	---	---	---	---	---

Или  $219_{10}$ .

Обычно побитовые операции применяются в тех случаях, когда необходимо выделить из двоичного представления числа содержимое некоторых его разрядов. Для этого применяют операцию *побитового И* между числом и специально подготовленным операндом, называемым **маской**. Маска содержит единицы в тех разрядах, содержимое которых необходимо выделить и нули в остальных. В дальнейшем результат операции может обрабатываться другими побитовыми операциями в соответствии с задачей. В программе удобно записывать значение маски восьмеричными или шестнадцатеричными константами.

Пример: *составить программу, обменивающую в коротком целом неотрицательном числе полубайты отдельно в младшем и старшем байтах.*



```
#include <iostream>
#include <clocale>
using namespace std;
int main() {
    setlocale(LC_CTYPE, "");
    unsigned short n, mask1, mask2, mask3, m1, st;
    cout << "Введите исходное число в 16-чной СС: ";
    cin >> hex >> n;
    //Маска для правого полубайта младшего байта:
    mask1 = 0xf;
    //Маска для левого полубайта младшего байта:
    mask2 = 0xf0;
```

```

//Маска для старшего байта:
mask3 = 0xff00;
//Следующий цикл выполнится два раза (по размеру переменной):
for (int i = 0; i < sizeof(n); i++) {
    //Выделяем правый полубайт преобразуемого байта:
    m1 = n & mask1;
    //Выделяем левый полубайт преобразуемого байта:
    st = n & mask2;
    //Объединяем противоположный байт и сдвинутые
    //выделенные полубайты:
    n = (n & mask3) | (m1 << 4) | (st >> 4);
    //Получаем маску для правого полубайта
    // следующего (более старшего) байта:
    mask1 <=< 8;
    //Получаем маску для левого полубайта
    // следующего (более старшего) байта:
    mask2 <=< 8;
    //Получаем маску для преобразованного (младшего) байта:
    mask3 >=> 8;
}
cout << "Преобразованное число = " << hex << n;
}

```

Фактически перечисленные выше действия по преобразованию числа можно заменить одним оператором:

```

n = ((n&0xf)<<4)    //Выделяем и сдвигаем правый полубайт
                    //младшего байта
| ((n&0xf0)>>4)    //Выделяем и сдвигаем левый полубайт
                    //младшего байта
| ((n&0xf00)<<4)    //Выделяем и сдвигаем правый полубайт
                    //старшего байта

```

`|((n&0xf000)>>4); //Выделяем и сдвигаем левый полубайт  
//младшего байта`

## СОДЕРЖАНИЕ РАБОТЫ

Выбрать алгоритм, составить его блок-схему и программу для решения выбранного варианта задания. Во всех вариантах предполагается, что размер короткого целого числа составляет 2 байта, а длинного – 4 байта. При необходимости ввод исходного значения и вывод результата может производиться в восьмеричном или шестнадцатеричном представлении.

## ВАРИАНТЫ ЗАДАНИЯ

1. Дано короткое целое неотрицательное число. Произвести сортировку его двоичного представления так, чтобы все единицы располагались в старших разрядах, а нули в младших.
2. Дано длинное целое неотрицательное число. Получить два коротких целых неотрицательных числа, одно из которых заполнено содержимым битов исходного числа с четными номерами, а другое – с нечетными.
3. Дано длинное целое неотрицательное число. Получить короткое целое неотрицательное число, биты которого начиная с младших содержат результат побитового И битов исходного числа соответственно с номерами 31 и 0, 30 и 1 и т.д.
4. Дано короткое целое неотрицательное число. Выполнить циклический сдвиг его двоичного представления на  $k$  битов влево.
5. Дано длинное целое неотрицательное число. Выполнить циклический сдвиг его шестнадцатеричного представления на  $k$  цифр вправо.

6. Дано короткое целое неотрицательное число. Определить, является ли его двоичное представление палиндромом.
7. Дано короткое целое неотрицательное число. Преобразовать старший байт числа таким образом, чтобы его двоичное представление стало палиндромом.
8. Дано короткое целое неотрицательное число. Определить в его двоичном представлении количество соседств двух нулей и двух единиц.
9. Дано длинное целое неотрицательное число. Определить количество различных цифр в его шестнадцатеричном представлении.
10. Дано короткое целое неотрицательное число. Сформировать другое число, которое будет состоять только из четных восьмеричных цифр исходного числа.
11. Дано длинное целое неотрицательное число. Заменить в его шестнадцатеричном представлении все нечетные цифры на нули.
12. Дано длинное целое неотрицательное число. Определить количество вхождений в него минимальной из цифр его шестнадцатеричного представления.
13. Дано короткое целое неотрицательное число. Произвести в его двоичном представлении обмен битов с номерами 0 и 1, 2 и 3, 4 и 5 и т.д.
14. Дано короткое целое неотрицательное число. Выполнить инверсию двоичного представления входящих в его состав четных восьмеричных цифр.
15. Дано длинное целое неотрицательное число. Удалить из его шестнадцатеричной записи цифры, меньшие 5.

16. Дано короткое целое неотрицательное число. Определить в его двоичном представлении максимальное количество расположенных рядом единиц.
17. Даны два коротких целых неотрицательных числа. Определить совпадают ли множества восьмеричных цифр, из которых они состоят.
18. Дано короткое целое неотрицательное число. Просматривая его двоичное представление парами бит, начиная с младших, произвести замену каждой комбинации «01» на «11», а «10» на «00».
19. Дано длинное целое неотрицательное число. Заменить в его шестнадцатеричном представлении старшую цифру максимальной из цифр числа, а младшую – минимальной.
20. Дано короткое целое неотрицательное число. Заменить каждую входящую в его состав шестнадцатеричную цифру на количество единиц, имеющихся в ее двоичном представлении (например, цифру F надо заменить на 4).
21. Дано короткое целое неотрицательное число. Определить в его двоичном представлении длину максимальной последовательности чередующихся нулей и единиц.
22. Дано короткое целое неотрицательное число. Заменить его старший бит результатом побитового исключающего ИЛИ всех предшествующих ему бит.
23. Дано длинное целое неотрицательное число, шестнадцатеричное представление которого не содержит ни одного нуля. Просматривая число, начиная с младших разрядов, оставить в нем только первые вхождения каждой цифры. Остальные вхождения заменить нулями.

24. Дано короткое целое неотрицательное число. Определить количество составляющих его шестнадцатеричных цифр, содержащих в двоичном представлении равное количество нулей и единиц (например, 3, 5 и др.).
25. Дано короткое целое неотрицательное число. Записать в две старшие цифры его шестнадцатеричного представления количество битов исходного числа, содержащих единицы.

### **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Перечислите побитовые операции, реализованные в языке C++ в порядке убывания приоритета.
2. В чем отличие операции побитового ИЛИ от операции арифметического сложения?
3. В чем отличие операций логического и побитового И?
4. Какие типы операндов допустимы в побитовых операциях?
5. Есть ли разница при выполнении операций сдвига целых знаковых и беззнаковых типов?
6. Существуют ли в языке C++ операции циклического сдвига?
7. Для чего, как правило, применяются побитовые операции?
8. Каким образом можно определить в целочисленной переменной значение некоторого одиночного бита или группы рядом стоящих бит?

## Лабораторная работа №9

# СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ STL.

## КЛАСС STRING. ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРНЫЕ КЛАССЫ

**Цель работы:** ознакомиться с концепцией построения стандартной библиотеки шаблонов STL; получить представление о возможностях класса *string*; приобрести навыки работы с последовательными контейнерными классами и алгоритмами их обработки.

### КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Стандартная библиотека шаблонов (Standard Template Library, STL) содержит набор классов коллекций (контейнерных классов) различного назначения и алгоритмов для работы с ними. В основе построения STL лежат идеи *обобщенного программирования*, которое использует шаблоны классов – некие параметризованные типы, которым можно передавать имя базового типа данных как аргумент. То есть целью обобщенного программирования является создание кода, который не зависит от конкретных типов данных, а позволяет оперировать терминами некоего обобщенного типа.

Основными компонентами STL являются:

- *Контейнеры* – структуры данных для управления коллекциями объектов определенного типа. Существует несколько разновидностей контейнеров, каждая из которых имеет свои достоинства и недостатки;
- *Итераторы* – специальные объекты, позволяющие перебирать элементы в контейнерах. Использование итераторов во многом похоже на использование указателей для перемещения по



элементам массива, а в целом они являются обобщением понятия указателя. В каждом контейнерном классе определен собственный тип итератора, учитывающий внутреннюю структуру контейнера;

- *Алгоритмы* – предназначены для обработки элементов контейнеров. Для всех их разновидностей определены некоторые наиболее часто используемые операции, такие как поиск элементов, их модификация, сортировка содержимого коллекции и т.д.

Прежде чем приступить к изучению указанных элементов, вкратце рассмотрим основные возможности строкового класса *string*, который непосредственно не входит в STL, но часто используется при решении различных задач средствами этой библиотеки.

## Класс *string*

В C++ нет стандартного строкового типа, встроенного в ядро языка. По умолчанию поддерживаются строки в стиле языка C, то есть массивы символов, текущая длина которых ограничивается нуль-символом (`'\0'`). Такой вид строки обеспечивает хорошее быстроедействие при ее обработке, но может иметь ряд потенциальных проблем, прежде всего в области безопасности работы программы. Обработка таких строк производится библиотечными функциями, заимствованными из языка C, прототипы которых определены в заголовочном файле *cstring*. Многие из них в своей работе ориентируются только на наличие нуль-символа как признака конца строки и формально не ограничивают количество обрабатываемых символов. То есть если в результате каких-то причин в массиве

символов, передаваемом такой функции, не окажется нуль-символа, то результат ее работы будет непредсказуемым.

Некоторым решением данной проблемы явилось добавление в библиотеку языка C функций с параметром, задающим максимальное количество обрабатываемых символов. При отсутствии нуль-символа в обрабатываемом массиве, такая функция прекращает свою работу после обработки указанного числа символов.

Другой проблемой является объявление самого массива символов. Если это статический массив, то при его объявлении указывается либо конкретное число элементов, либо оно определяется в ходе инициализации. Расширение строки за границы такого массива невозможно. Если массив динамический, то расширение его возможно, однако это потребует от программиста дополнительных усилий.

Таким образом, требовался тип, представляющий строку не как коллекцию отдельных символов, а как единую сущность, предоставляющий возможности ее прозрачного для программиста динамического расширения и инкапсулирующий в себе некоторые функции ее обработки в соответствии с парадигмой объектно-ориентированного программирования. Поэтому в состав стандартной библиотеки языка C++ был добавлен строковый класс *string*, который описан в одноименном заголовочном файле *string*.

В основе строкового класса лежит шаблонный класс *basic\_string*, с помощью которого определяются несколько разновидностей строк, объявляемых через инструкцию *typedef* для разных символьных типов. Далее будет рассматриваться непосредственно класс *string*, который определен для типа элементов *char*. Для других базовых типов нужно использовать другие строковые классы, например, *wstring* для расширенного символьного типа *wchar\_t*. Все рассмотренные ниже

возможности класса *string* идентичны возможностям других строковых классов (с поправкой на базовый тип). И все строковые классы входят в пространство имен *std*, поэтому либо их имена нужно использовать с префиксом *std::*, либо предварительно размещать директиву *using*.

Экземпляр класса *string* создается с помощью одного из конструкторов. Создавать можно пустой объект (применяется конструктор по умолчанию), строку на основе другого экземпляра класса *string* (с помощью конструктора копирования), заполненную заданным символом и заранее определенной длины или на основе строки в стиле языка C с нуль-символом в конце (с помощью конструкторов с различным набором параметров). Ниже показаны несколько примеров создания экземпляров класса *string*:

```
//Пустая строка:
std::string str1;

//Строка с содержимым «Мама мыла»:
std::string str2("Мама мыла");
char s[] = "Hello!";
//Строка с содержимым «Hello»:
std::string str3(s, 5);
//Строка с содержимым «Мама мыла»:
std::string str4(str2);
//Строка с содержимым «Мама»:
std::string str5(str4, 4);
//Строка с содержимым «AAAAA»:
std::string str6(5, 'A');
```

Созданный такими способами объект класса *string* не является константным и может быть изменен. Для этого могут использоваться операции присваивания, средства консольного или файлового ввода, а

также различные методы, определенные в классе *string*. Далее вкратце будут рассмотрены все варианты.

Для класса *string* перегружена операция присваивания с тремя типами операндов. Присвоить можно другую строку класса *string*, строку в стиле языка C с нуль-символом в конце или одиночный символ (выражение типа *char*), например:

```
std::string str7;  
str7 = "ABC";  
std::string str8 = str7;  
str7 = 'a';
```

Для класса *string* перегружен и ряд других операций. В частности, строки можно сравнивать между собой с помощью операций отношения (сравнение происходит в соответствии с ASCII кодами символов, входящих в строки). Причем сравнивать можно как два экземпляра класса *string*, так и экземпляр со строкой в стиле языка C. Например, для объявленных выше переменных следующие два оператора выведут в консоль одно и то же (строку *"ABC" меньше "a"*):

```
if (str8 < str7) std::cout << "\"ABC\" меньше \"a\"<<'\\n';  
if (str8 < "a") std::cout << "\"ABC\" меньше \"a\"<<'\\n';
```

Для доступа к отдельным символам строки можно использовать операцию индексации «[ ]». Например, выражение *str8[0]* вернет символ 'A'.

По сравнению со строками в стиле языка C, класс *string* позволяет гораздо проще осуществлять конкатенацию строк. Для этого в классе определены операции «+» и «+=». Применять их можно и к экземплярам класса, и к строкам с нуль-символом в конце, но результат должен присваиваться переменной типа *string*, например:

```
std::string str9 = "Мама";
```

```
str9 += " мыла ";  
str9 = str9 + "раму";
```

В результате в переменной *str9* окажется содержимое «Мама мыла раму».

Для ввода-вывода переменных класса *string* можно использовать рассмотренные в предыдущих работах средства консольного и файлового ввода-вывода. Операции помещения в поток и извлечения из потока перегружены в том числе и для типа *string* и работают также, как и в случае строк с нуль-символом в конце, например:

```
std::cout << str9 << "\n"; // Вывод в консоль «Мама мыла раму»  
std::cin >> str9; // Ввод одной лексемы
```

Вывод в целом работает аналогично строкам с нуль-символом в конце. При вводе размер переменной строкового класса динамически расширяется при необходимости. Операция извлечения из потока «>>», также, как и в случае строки в стиле языка C, позволит ввести одну лексему, ограниченную символами пробела, табуляции или конца строки. Для ввода строк с нуль-символом, содержащих пробелы и/или символы табуляции, ранее рассматривались методы *get()* и *getline()* класса *istream* и его потомков. В случае же строк класса *string* нужно применять функцию (не метод!) *getline()*, объявленную в заголовочном файле *string*. Функции в качестве параметров передаются входной поток, переменная класса *string* и как необязательный параметр — символ-разделитель (по умолчанию символ новой строки '\n', разделитель из потока удаляется, но не добавляется в строку). Например:

```
// Ввод строки с клавиатуры, пока во входном потоке не встретится '\n':  
std::getline(std::cin, str9);
```

```
// Ввод строки с клавиатуры, пока во входном потоке не встретится ';' :  
std::getline(std::cin, str9, ';');
```

Для обработки содержимого строки в классе *string* имеется большое количество разнообразных методов. Далее вкратце будут рассмотрены некоторые из них.

Для определения текущей длины строки в классе *string* имеется два одинаково работающих метода: *size()* и *length()*. Пример:

```
std::string str_src("Здравствуй, мир!"); str_dest;  
int len;  
len = str_src.size(); // Переменная len равна 16  
len = str_dest.length(); // Переменная len равна 0
```

Присвоить часть одной строки другой можно с помощью метода *assign()*. Исходная строка может быть как переменной класса *string*, так и строкой с нуль-символом в конце. Строка-источник может быть присвоена строке-результату полностью или частично, в виде набора символов с заданной позиции (отсчет с нуля) и заданного количества (данный метод, как и большинство других в классе *string*, имеют несколько перегруженных вариантов с разным набором параметров, выбирается тот, который наиболее подходит для реализации конкретной задачи). Например:

```
str_dest.assign(str_src, 12, 4); // Присвоить str_dest "мир!"
```

Для вставки части одной строки в другую используется метод *insert()*. Метод перегружен для разного набора параметров, в самом общем случае происходит вставка в вызывающую строку с заданной позиции части исходной строки, заданной размерности и начиная с указанной позиции. Также можно вставлять строку с нуль-символом в конце, указывая количество вставляемых символов. Например:

```
str_src = "Петя";
```

```
str_dest = "Вася";  
str_dest.insert(4, str_src, 0, 4); //Результат: "ВасяПетя"  
str_dest.insert(4, " ", 2); //Результат: "Вася, Петя"
```

Удаление части строки производится с помощью метода *erase()*. При указании в скобках стартовой позиции и количества удаляемых символов производится частичная очистка строки. Для полной очистки строки можно вызвать метод *clear()*. Например:

```
str_dest.erase(4, 6); //Результат: "Вася, Петя" -> "Вася"  
str_dest.clear(); //Результат: пустая строка
```

Замена части строки на часть другой строки производится методом *replace()*. Метод также перегружен для разных наборов параметров, в общем случае указывается стартовая позиция и количество заменяемых символов в вызывающей строке, строка-источник, стартовая позиция в ней и количество символов для замены, например:

```
str_src = "Тетя чистила картошку";  
str_dest = "Мама мыла раму";  
// Получаем в str_dest строку "Мама чистила раму":  
str_dest.replace(5, 4, str_src, 5, 7);
```

Для выделения части строки с заданной позиции и заданного размера используется метод *substr()*. Метод возвращает значение типа *string*. Пример:

```
str_src = "Здравствуй, мир!";  
str_dest = str_src.substr(12, 3); //В str_dest строка "мир"
```

Осуществить поиск подстроки в строке можно с помощью базового метода *find()*. Метод перегружен в нескольких вариантах с различными наборами параметров и ищет первое (слева) вхождение строки класса *string*, строки с нуль-символов в конце или отдельного символа начиная с заданной позиции вызывающей строки. Метод возвращает позицию

(индекс) стартового символа найденной подстроки (символа) или -1, если поиск завершился неудачно. Примеры:

```
str_src = "Мама мыла раму";
str_dest = "мыла";
int pos;
pos = str_src.find(str_dest, 0); //Результат: 5
pos = str_src.find(str_dest, 10); //Результат: -1
pos = str_src.find("раму", 5); //Результат: 10
```

Существуют также и другие методы для поиска, имеющие те же наборы параметров. Например, метод *rfind()* ищет самое последнее (то есть первое справа) вхождение подстроки или символа, а методы *find\_first\_of()* и *find\_last\_of()* ищут, соответственно, первое и последнее вхождение любого символа искомой подстроки в вызывающую строку.

В ряде случаев, оперируя в программе строкой класса *string*, нужно воспользоваться какими-либо из библиотечных функций, принимающими в качестве параметра строку с нуль-символом в конце. Для этого используется метод *c\_str()*, возвращающий константный указатель на строку с нуль-символом в конце. Менять содержимое полученной строки нельзя. В примере ниже с помощью функции *strlen*, объявленной в заголовочном файле *cstring* и определяющей длину строки в стиле языка C, вычисляется размер возвращенной методом *c\_str()* строки с нуль-символом в конце:

```
str_src = "Здравствуй, мир!";
int len = strlen(str_src.c_str()); //Переменная len равна 16
```

Достаточно часто возникает необходимость конвертации числового значения в строку. С этой целью в стандарте языка C++11 была представлена функция *to\_string()*, объявленная в заголовочном файле *string*. Функция имеет один параметр числового типа и возвращает



переменную класса *string*. Она перегружена для всех основных числовых типов. При преобразовании вещественных значений дробная часть записывается шестью знаками. Примеры применения функции:

```
str_dest = std::to_string(24.57); //Результат: "24.570000"  
str_dest = std::to_string(489); //Результат: "489"
```

Обратное преобразование переменной класса *string* в числовой тип может производиться рядом функций с разными результирующими типами, также введенными в стандарте C++11. В качестве примера рассмотрим преобразование строки в тип *int* с помощью функции *stoi()* и в тип *double* с помощью функции *stod()*. В строке разделителем целой и дробной части должна быть точка. Причем если в строке представлено число с дробной частью, то оно может быть преобразовано функцией *stoi()* в целое путем отбрасывания дробной части. Примеры использования:

```
str_src = "631";  
int i = std::stoi(str_src); //Результат: 631  
str_src = "321.569";  
i = std::stoi(str_src); //Результат: 321  
double d = std::stod(str_src); //Результат: 321.569
```

## Контейнерный класс *vector*

Шаблонный класс *vector* представляет собой пример *последовательного* контейнера с возможностью *произвольного доступа*. Особенностью последовательных контейнеров является то, что позиция элемента в коллекции зависит от времени и места вставки, но не зависит от его значения (в отличие от *ассоциативного* контейнера, где содержимое сортируется в соответствии со значениями элементов по заданному критерию сортировки). Внутри вектор представляет

собой (как правило, хотя стандартом это не фиксируется) динамический массив, размерность которого может меняться по мере добавления или удаления элементов. От обычного динамического массива вектор отличается тем, что напрямую программистом операции управления динамической памятью *new* и *delete* не используются, а их применение производится автоматически, при выполнении методов класса, влияющих на размерность вектора.

Как и все элементы библиотеки STL, класс *vector* входит в стандартное пространство имен *std*, поэтому либо его имя нужно использовать с префиксом *std::*, либо предварительно размещать директиву *using*. Сам класс описан в заголовочном файле *vector*, который нужно включать в текст программы с помощью директивы *#include*.

Объявить в программе можно изначально пустой вектор или же сразу задать начальную его размерность, например:

```
std::vector<int> vc1; //Целочисленный вектор нулевого размера  
std::vector<double> vc2(10); //Вещественный вектор из 10 элем.
```

Для создания переменной *vc1* вызывается конструктор класса по умолчанию (без параметров). Для переменной *vc2* применяется конструктор с параметром, определяющим первоначальную емкость контейнера, в данном случае 10 элементов. Текущую емкость вектора можно узнать, вызвав метод *capacity()*. Реальный же размер обоих векторов (то есть количество фактически добавленных в вектор элементов), который можно определить с помощью стандартного для контейнеров STL метода *size()*, после таких объявлений равен нулю. Элементы в эти векторы могут добавляться с помощью соответствующих методов класса, которые будут рассмотрены ниже. При этом вектор *vc2* гарантированно будет расширяться только после

добавления десяти элементов. Поскольку изменение емкости вектора является достаточно дорогостоящей с точки зрения алгоритмической сложности операцией, то при объявлении переменной, если примерная необходимая емкость заранее известна, лучше ее указывать.

Базовый тип вектора указывается в угловых скобках после слова *vector* (так как все контейнеры являются шаблонными классами). В качестве базового типа может выступать любой простой или составной тип, включая пользовательские типы, объявленные классы и т.д., которые допустимо применять в шаблонных классах. В том числе, в качестве базового типа может использоваться другой контейнер. Таким образом, к примеру, можно объявить вектор векторов, то есть вектор, каждый элемент которого представляет собой другой вектор. Ниже показан пример, где объявляется такой вектор и с помощью инициализации заполняется начальными значениями:

```
std::vector<std::vector<int>> vc3{{1, 2, 3}, {4, 5, 6}};
```

Получается вектор, состоящий на момент инициализации из двух целочисленных векторов, каждый из которых состоит из трех элементов. При этом память для инициализирующих значений уже выделена, можно обращаться к этим элементам, изменять их при необходимости. Также в указанный вектор в дальнейшем можно добавлять любое количество целочисленных векторов, каждый из которых может содержать произвольное количество элементов. В том числе элементы можно добавлять и в те целочисленные векторы, которые уже заданы при инициализации. Безусловно, такие действия могут потребовать изменения емкости вектора.

Для векторов определен ряд операций отношения: «<», «<=», «==», «!=», а также операция присваивания «=». Сравнение и присваивание одного другому происходит только для векторов с одинаковым базовым

типом. Один вектор меньше другого, если первый из различающихся элементов меньше, чем соответствующий в другом векторе. Векторы равны, если они имеют одинаковый размер и у них совпадают все пары элементов. Присваивание приводит к замещению в векторе слева от операции « $\Rightarrow$ » старых элементов на новые и контейнеры становятся идентичными.

Для обращения к элементам вектора определена операция индексации « $[ ]$ », которая работает также, как и в случае массивов. Например, к первому элементу второго целочисленного вектора из предыдущего примера (со значением 4) можно обратиться так: `vc3[1][0]`. Для некоторых контейнерных классов, включая и вектор, определен специальный метод доступа к элементам `at()`. Отличие его от операции индексации состоит в том, что в случае указания некорректного индекса генерируется исключение класса `out_of_range`, которое может быть обработано. Однако быстроедействие такого метода существенно ниже. Например, вывести в консоль такой же элемент вектора с помощью данного метода можно так:

```
std::cout << vc3.at(1).at(0); //Результат: вывод в консоль 4
```

Кроме того, имеются два отдельных метода, возвращающие первый и последний элемент: `front()` и `back()`. Вывести в консоль такой же элемент, как показано выше, можно с помощью метода `front()`:

```
std::cout << vc3[1].front();
```

Изменять размер вектора можно добавляя в него новые элементы, удаляя их по отдельности или группой. Также имеется метод `resize()`, который позволяет увеличивать или уменьшать вектор до заданной размерности, указываемой в качестве первого параметра. Второй, опциональный параметр содержит значение, которым заполняются добавляемые в вектор элементы. Если новая размерность меньше

текущей, «лишние» элементы в конце вектора удаляются. Например, для ранее объявленного вектора *vc1*:

```
vc1.resize(20, 1); //Расширить до 20 элем., заполнить единицами  
vc1.resize(5); //Уменьшить до 5 элементов
```

Добавить в конец вектора новый элемент базового типа можно с помощью метода *push\_back()*. В качестве единственного параметра метод принимает значение базового типа, которое получает добавленный элемент. Например, для объявленных выше векторов:

```
vc1.push_back(10); //Добавить один элемент со значением 10  
//Добавить один целочисленный вектор, используется значение,  
//возвращаемое конструктором по умолчанию класса vector:  
vc3.push_back(std::vector<int>());  
//В добавленном векторе создать пять элементов, заполнить нулями:  
for(int i = 0; i < 5; i++)  
    vc3.back().push_back(0);
```

Удаление одного элемента в конце вектора производится методом *pop\_back()*. Например:

```
vc1.pop_back();
```

Перед рассмотрением других методов класса *vector* необходимо более подробно остановиться на упомянутом выше понятии *итератора*. Итератор обобщает понятие указателя и используется для того, чтобы работать с различными структурами данных стандартным способом. Основным в итераторах является то, что они обеспечивают перемещение по записям контейнера и обращение к ним. Имеются различные виды итераторов, которые налагают определенные ограничения на способы перехода по записям (однонаправленный, двунаправленный, произвольный) и доступа к ним (чтение, запись).

Класс *vector* подразумевает использование итераторов произвольного доступа, обладающих максимальными возможностями.

В C++ имеется общий класс *iterator*, однако, чаще всего для объявления переменной-итератора используется не он, а класс итератора, включенный в состав соответствующего контейнерного класса. Например, объявление итератора для работы с целочисленным вектором может выглядеть так:

```
std::vector<int>::iterator iter;
```

Для итератора определен ряд операций, поддерживаемый набор которых зависит от его типа. С рассматриваемым в данном случае итератором произвольного доступа могут использоваться следующие операции:

- «+», «-» (используются для построения адресных выражений, например, *iter+3*, задающих расположение нужного элемента вектора относительно текущего значения итератора);
- «+=», «-=» (аналогично предыдущему пункту, но с изменением значения самого итератора);
- «++», «--» (префиксный и постфиксный инкремент и декремент, обеспечивают переход итератора к следующему или предыдущему элементу);
- «\*» (разыменование, предоставляющее доступ через итератор к элементу вектора);
- «<», «<=», «>», «>=», «==», «!=» (операции отношения для сравнения двух итераторов);
- «[ ]» (индексация, работает также, как и при непосредственном применении к вектору).

Итератор чаще всего используется для прохождения всех имеющихся в векторе элементов, для чего он в цикле инициализируется

ссылкой на первый элемент, а затем на каждом шаге производится проверка выхода итератора за границы вектора и увеличение на единицу (увеличивать необязательно на единицу, все определяется задачей). В теле цикла производятся требуемые действия с элементами вектора, применяя разыменование итератора.

Для получения граничных значений итератора в классе *vector* определены два метода: *begin()* и *end()*. Первый метод возвращает значение итератора, указывающего на первый элемент вектора. Второй – возвращает итератор, указывающий на элемент, следующий за последним элементом. Поскольку такого элемента как бы не существует, то применение разыменования к такому значению итератора приведет к возникновению ошибки. Но проверять текущее значение итератора на совпадение с значением, возвращенным методом *end()* можно и в случае их равенства осуществляется выход из цикла.

Выше был показан пример объявления переменной-итератора для работы с вектором. Избавиться от необходимости указания подобного громоздкого наименования типа итератора можно, если применить появившийся в стандарте C++11 механизм вывода типа переменной из инициализирующего ее значения. Для этой цели вместо типа указывается ключевое слово *auto*, которое в предыдущих версиях использовалось для явного указания класса автоматических переменных. Например, вывод в консоль элементов объявленного выше вектора *vc1* может выглядеть так:

```
for(auto i = vc1.begin(); i != vc1.end(); i++)  
    std::cout << *i << ' ';
```

Другой возможностью простого перебора всех элементов вектора в прямом направлении является использование цикла *for*, основанного на диапазоне. Иначе называемый циклом в стиле *for-each*, данный вариант

применения оператора *for* появился также в стандарте C++11. В отличие от обычного варианта, в скобках вместо трех выражений, разделенных точкой с запятой, размещается одно, где объявляется управляющая переменная базового типа вектора и через двоеточие указывается сам вектор, элементы которого нужно перебрать. В результате на каждом шаге управляющая переменная получает значение очередного элемента вектора, которое можно обрабатывать в теле цикла. Возможно автоматическое выведение типа управляющей переменной при указании ключевого слова *auto*. Например, вывод в консоль элементов объявленного выше вектора *vc3* может выглядеть следующим образом:

```
for (auto i : vc3) {  
    for (auto j : i)  
        std::cout << j << ' '  
    std::cout << '\n';  
}
```

В данном примере переменная *i* получит автоматически выведенный тип целочисленного вектора, а переменная *j* будет иметь тип его элементов *int*. В отличие от случая с использованием итератора, здесь разыменование переменной *j* в теле цикла не производится.

Рассмотренный вариант цикла позволяет получать доступ к элементам вектора только в режиме чтения. Если необходимо иметь возможность в цикле изменять содержимое вектора, то в качестве управляющей требуется использовать ссылочную переменную, указывая перед ее именем символ амперсанда «&». Следующий пример показывает изменение в цикле элементов вектора *vc1* и вывод измененных значений в консоль:

```
for (auto &i : vc1) {  
    i += 10;
```



```
std::cout << i << ' ';
}
```

Перемещение по вектору можно организовать и в обратном направлении, от последнего элемента к первому. Проще всего для этой цели применить так называемый обратный итератор класса *reverse\_iterator*, который также включается в состав класса *vector*. Для работы с таким итератором в его составе определены два метода: *rbegin()* и *rend()*. Первый метод возвращает значение, указывающее на первый элемент в обратной последовательности. По сути же разыменованное такое значение итератора дает доступ к последнему элементу вектора. Второй метод возвращает значение итератора, указывающего на элемент, следующий за последним в обратной последовательности. По сути, это ссылка на несуществующий элемент перед первым элементом вектора. По достижении итератором такого значения, осуществляется выход из цикла. Также нужно учесть, что на каждом шаге цикла текущее значение итератора увеличивается, хотя в действительности это приводит к обратному перемещению по вектору. Пример обратного перебора вектора *vc1*, где в цикле каждый элемент увеличивается на единицу и выводится в консоль приведен ниже:

```
for (auto i = vc1.rbegin(); i != vc1.rend(); i++)
    std::cout << ++(*i) << ' ';
```

Теперь рассмотрим ряд методов класса *vector*, которые позволяют управлять изменением составляющих его объектов. Часть таких методов используют итераторы в качестве параметров, определяя с их помощью позиции обрабатываемых элементов.

Метод *insert()* позволяет вставить в вектор один или несколько новых элементов начиная с заданной позиции, которая определяется значением первого параметра, являющегося итератором. В разных

перегруженных вариантах метода количество и тип последующих параметров различаются. Это может быть значение базового типа, количество раз, сколько его нужно вставить или диапазон итераторов из другого контейнера, который выступает в качестве источника данных. В последнем варианте базовые типы целевого контейнера и контейнера-источника не должны строго совпадать. Достаточно, чтобы элементы, извлеченные из источника, могли быть приведены к базовому типу целевого вектора. Ниже приведены примеры использования разных вариантов данного метода для ранее объявленных векторов:

```
//Вставить в вектор vc1 два элемента со значением 5 со второй позиции:  
vc1.insert(vc1.begin()+1, 2, 5);  
  
//Вставить в конце вектора vc1 все элементы первого из целочисленных  
//векторов, входящих в состав вектора vc3:  
vc1.insert(vc1.end(), vc3[0].begin(), vc3[0].end());
```

Нужно обратить внимание, что в последнем примере третий параметр имеет значение итератора, указывающего не на последний элемент контейнера-источника, подлежащий вставке, а на элемент, стоящий непосредственно за ним (в общем-то несуществующий). Это нужно учитывать при построении адресных выражений, которые могут использоваться в качестве такого параметра метода.

Также нужно учесть, что если вставка элементов приводит к превышению текущей емкости вектора и необходимости его расширения, то все полученные ранее значения итераторов становятся недействительными и требуется их повторное вычисление.

Метод *erase()* позволяет удалить выборочно один элемент вектора или группу смежных элементов. В первом случае метод имеет один параметр со значением итератора, указывающего на удаляемый элемент. Во втором – два параметра-итератора, определяющих первый

удаляемый элемент и следующий за последним удаляемым элементом.

Например:

```
//Удалить в векторе vc1 второй элемент:  
vc1.erase(vc1.begin()+1);  
  
//Удалить в векторе vc1 первый и второй элементы:  
vc1.erase(vc1.begin(), vc1.begin()+2);
```

Метод *swap()* позволяет произвести обмен элементов двух векторов одного базового типа, но необязательно одинаковой текущей размерности. Например, обмен элементами вектора *vc1* и первого из целочисленных векторов из состава вектора *vc3* выглядит так:

```
vc1.swap(vc3[0]); //Аналогично: vc3[0].swap(vc1);
```

## Контейнерный класс *list*

Шаблонный класс *list* является еще одним примером последовательного контейнера, но без возможности произвольного доступа. Данный контейнер основан на двухсвязном линейном списке, в котором элементы кроме самого значения базового типа содержат два указателя на предыдущий и следующий элементы. Это значит, что в отличие от класса *vector*, в классе *list* расширение контейнера происходит при каждом добавлении нового элемента, что требует всегда одинакового времени и выполняется быстро, также, как и удаление элемента. Но при этом перемещение к *n*-му элементу списка от начала будет требовать времени, пропорционального *n*, в отличие от класса *vector* с его произвольным доступом к элементам.

Для использования класса *list* нужно включить в текст программы с помощью директивы препроцессора *#include* заголовочный файл *list*. Так же, как и класс *vector*, класс *list* входит в стандартное пространство имен *std* и с этой точки зрения его использование аналогично вектору.

Класс *list* поддерживает тот же набор конструкторов, что и класс *vector*. Несколько примеров использования конструкторов разных видов:

```
std::list<int> lst1; //Целочисленный пустой список
std::list<double> lst2(10); //Веществ. список из 10 нулевых элем.
std::list<int> lst3(5, 3); //Целочисленный список из пяти троек
std::list<int> lst4(lst3); //Создание lst4 как копии lst3
//Значения берутся из списка инициализации:
std::list<int> lst5({ 1,2,3,4,5 });
```

Как и в случае вектора, в классе *list* значения, полученные при начальной инициализации, могут в дальнейшем изменяться, удаляться, содержимое контейнера может дополняться новыми элементами.

В классе *list* определены те же операции отношения и присваивания, что и в классе *vector*, и работают они аналогично. Из-за невозможности произвольного доступа к элементам списка, операция индексации «[ ]» в данном классе не определена. Доступ к элементам списка может осуществляться только последовательно, от элемента к элементу, в прямом или обратном направлении.

В отличие от вектора, в классе *list* не определен метод *capacity()*, так как при добавлении нового элемента всегда происходит выделение для него динамической памяти. Но также, как и в векторе, текущий размер списка можно изменить с помощью метода *resize()*, например:

```
lst1.resize(15, 1); //Теперь это список из пятнадцати единиц
```

Также как и в классе *vector*, в *list* можно получить значение первого и последнего элементов с помощью методов *front()* и *back()* соответственно. Для доступа к остальным элементам списка можно использовать итератор. Прямой итератор можно получить используя методы *begin()* и *end()*. Традиционно, первый метод возвращает

итератор, указывающий на первый элемент списка, а второй – на элемент, следующий за последним. Например, вывод списка в консоль можно организовать следующим образом:

```
for (auto i = lst5.begin(); i != lst5.end(); i++)
    std::cout << *i << ' ';
```

Аналогичным образом можно осуществлять и перемещение по списку в обратном порядке, так как он является двухсвязным. Для этой цели применяются такие же обратные итераторы, что и в случае применения вектора и значения которых возвращаются при вызове тех же методов *rbegin()* и *rend()*. Например, список *lst5* вывести в консоль в обратном направлении можно так:

```
for (auto i = lst5.rbegin(); i != lst5.rend(); i++)
    std::cout << *i << ' ';
```

Подобно векторам, перебор элементов списка в прямом направлении можно осуществлять и с помощью цикла в стиле *for-each*. В следующем примере каждый элемент списка увеличивается на 50 и выводится в консоль:

```
for (auto &i : lst5) {
    i += 10;
    std::cout << i << ' ';
}
```

Класс *list* позволяет добавлять элементы как с начала списка, так и с конца. Для этой цели определены методы *push\_front()* и *push\_back()*, которые работают аналогично рассмотренному выше методу *push\_back()* в классе *vector*. Удаление первого и последнего элементов списка можно осуществить методами *pop\_front()* и *pop\_back()*.

Особенностью класса *list* является то, что после вставки и/или удаления элементов в контейнере все определенные на этот момент

итераторы остаются действительными. Это связано с принципом функционирования двухсвязных списков, в которых когда-либо добавленный элемент всегда остается на месте (если он не удаляется, конечно) при выполнении таких операций, а корректируются только указатели, связывающие элементы в логическую последовательность.

Для управления содержимым контейнера имеется ряд методов, аналогичных классу *vector*: *insert()*, *erase()*, *swap()*, *clear()*. Дополнительно имеется ряд методов, характерных именно для данного типа контейнера, основанного на структуре двухсвязного списка. Метод *reverse()* меняет порядок следования элементов списка на обратный. Пример вызова:

```
lst5.reverse();
```

Метод *sort()* сортирует элементы списка. При использовании метода без параметров сортировка всегда происходит в порядке возрастания. Если же требуется иной порядок сортировки и/или в качестве базового в контейнере используется пользовательский тип, применяется второй перегруженный вариант метода, имеющий в качестве параметра функциональный объект *Compare*. В качестве такового проще всего использовать указатель на функцию, принимающую в качестве параметров две константные ссылки на переменные базового типа и возвращающую значение типа *bool*. Соответственно, перед вызовом метода *sort()* данная функция должна быть определена, или, как минимум, объявлен ее прототип. Ниже показан пример сортировки списка *lst5* в порядке невозрастания с использованием предварительно определенной функции *cmp()*, осуществляющей сравнение двух элементов контейнера:

```
bool cmp(const int& val1, const int& val2) {  
    return val1 >= val2;  
}
```

```
}
```

```
...
```

```
lst5.sort(cmp);
```

Метод *remove()* позволяет удалить из списка все элементы с заданным значением. Пример:

```
lst5.remove(3); //Удаляет из списка элемент со значением 3
```

Метод *unique()* при наличии в списке серии подряд идущих одинаковых элементов оставляет только первый элемент. Например:

```
lst1.unique(); //Список из пятнадцати единиц сокращается до одной
```

Метод *splice()* сцепляет два списка, производя эту операцию без перераспределения динамической памяти, только лишь за счет коррекции указателей. Метод имеет несколько перегруженных вариантов, где в базовом в качестве параметров задаются итератор, ссылающийся на позицию, перед которой будет вставлено содержимое списка-источника, и, собственно, ссылка на этот источник. Во втором варианте добавляется еще один параметр-итератор, указывающий на единичный элемент списка-источника, который будет вставлен перед элементом, заданным первым параметром. В третьем варианте добавляются не один, а два параметра-итератора, задающих начало и конец вставляемого диапазона элементов из списка-источника. В первом и третьем варианте список-источник обязательно должен отличаться от списка-цели и после вызова метода источник остается пустым. Во втором варианте допускается вставлять в указанную позицию единичный элемент из этого же списка. Примеры:

```
std::list<int> lst6({ 10, 20, 30 });
```

```
std::list<int> lst7({ 40, 50 });
```

```
//В lst6 результат сцепления: 10, 20, 30, 40, 50:
```

```
lst6.splice(lst6.end(), lst7);
```

```

//В lst6 на 2-ю позицию перемещен последний элемент: 10, 50, 20, 30, 40:
lst6.splice(++lst6.begin(), lst6, --lst6.end());
lst7 = lst6; //Копирование списка
//В конце lst6 добавлены элементы со второго по предпоследний
// из списка lst7, результат: 10, 50, 20, 30, 40, 50, 20, 30:
lst6.splice(lst6.end(), lst7, ++lst7.begin(),
            --lst7.end());

```

Объединение двух списков, требующее перераспределение динамической памяти, производится методом *merge()*. Оба списка должны быть упорядочены одинаковым образом. Результат объединения также будет упорядочен. В случае, если списки упорядочены по возрастанию, то метод применяется в варианте с одним параметром – ссылкой на список-источник. Причем после объединения источник также остается пустым. Если же списки были упорядочены другим способом, то применяется вариант с дополнительным параметром, в качестве которого выступает ссылка на функциональный объект *Compare*, такой же, как применялся в рассмотренном выше методе *sort()*. Покажем как производится сортировка двух списков в порядке невозрастания с помощью определенной в приведенном выше примере функции сравнения *cmp()* с последующим их объединением вызовом метода *merge()*:

```

std::list<int> lst8({ 10, 30 });
std::list<int> lst9({ 20, 40 });
lst8.sort(cmp);
lst9.sort(cmp);
lst8.merge(lst9, cmp); //Результат в lst8: 40, 30, 20, 10

```



## Алгоритмы

Алгоритмы представляют собой унифицированное решение по обработке содержимого контейнеров различных видов. В рассмотренных выше примерах контейнерных классов практически не были представлены встроенные методы, предоставляющие возможности по подобной обработке. Это связано с тем, что данный функционал вынесен в отдельную библиотеку автономных функций, которые работают с контейнерами разных видов как с шаблонами классов. Такой подход делает механизм обработки содержимого контейнеров более мощным и гибким, сокращая при этом объем программного кода.

Все алгоритмы делятся на несколько категорий, из которых в данной работе вкратце будут рассмотрены следующие:

- немодифицирующие операции с последовательностями;
- модифицирующие операции с последовательностями;
- операции, связанные с сортировкой последовательностей.

Для применения данных алгоритмов необходимо в текст программы включить заголовочный файл *algorithm*. Общим для всех функций, реализующих алгоритмы, является то, что границы обрабатываемых диапазонов задаются с помощью итераторов. Если функции имеют возвращаемое значение, то оно, как правило, также является итератором. Кроме итераторов, такие функции могут содержать параметр, являющийся функциональным объектом, аналогичным рассмотренному ранее *Compare* в методе сортировки списка. Такой объект может применяться в процессе работы алгоритма к каждому элементу последовательности или к группам элементам. Упомянутый

*Compare*, к примеру, играл в процессе сортировки списка роль бинарного *предиката*, используемого для сравнения двух элементов.

В качестве функциональных объектов могут применяться экземпляры классов, у которых имеется перегруженная операция «( )» или обычные функции и указатели на функции (как в упомянутом выше примере). В зависимости от количества аргументов такие функциональные объекты делятся на три вида:

- *генератор* – функциональный объект без аргументов;
- *унарная функция* – функциональный объект с одним аргументом;
- *бинарная функция* – функциональный объект с двумя аргументами.

Отдельно нужно упомянуть два случая. Унарная функция, возвращающая значение типа *bool*, представляет собой предикат (унарный). Бинарная функция, возвращающая значение типа *bool*, представляет собой бинарный предикат.

Во многих случаях к элементам контейнера одного из базовых типов (но не составных пользовательских!) нужно применить какую-либо арифметическую или логическую операцию (как, опять-таки в рассмотренном ранее примере с сортировкой). Для того чтобы каждый раз не объявлять такую функцию, в STL имеется набор предопределенных функциональных объектов, соответствующих основным арифметическим и логическим операциям. Данные функциональные объекты определены в заголовочном файле *functional* и представлены в следующей таблице:

Операция	Функциональный объект	Тип объекта	Результат
+	plus	бинарный	x+y
-	minus	бинарный	x-y

Операция	Функциональный объект	Тип объекта	Результат
*	multiplies	бинарный	$x*y$
/	divides	бинарный	$x/y$
%	modulus	бинарный	$x\%y$
-	negate	унарный	$-x$
==	equal_to	бинарный	$x==y$
!=	not_equal_to	бинарный	$x!=y$
>	greater	бинарный	$x>y$
<	less	бинарный	$x<y$
>=	greater_equal	бинарный	$x>=y$
<=	less_equal	бинарный	$x<=y$
&&	logical_and	бинарный	$x\&y$
	logical_or	бинарный	$x  y$
!	logical_not	унарный	$!x$

Для применения предопределенного функционального объекта при вызове одной из функций-алгоритмов нужно указать его имя и в угловых скобках тип элементов контейнера, например: *plus<double>( )* или *less\_equal<int>( )*.

Далее рассмотрим несколько функций-алгоритмов из каждой упомянутой выше категории. С полным перечнем всех алгоритмов можно ознакомиться в стандарте языка (например, в <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf>). В приведенных ниже примерах вызова будем использовать следующие векторы:

```
std::vector<int> vec_1st{1, 5, 2, 8, 1, 4, 9, 6, 6};
std::vector<int> vec_2nd{8, 1, 4};
```

Однако в абсолютном большинстве случаев рассмотренные алгоритмы применимы и к последовательностям класса *list*.

К числу немодифицирующих операций с последовательностями относятся:

- *adjacent\_find()* – алгоритм выполняет поиск пары соседних значений в контейнере. В первом варианте принимает два параметра-итератора, задающие диапазон поиска, возвращает итератор на первый элемент из пары соседних одинаковых значений или на элемент, следующий за последним, если поиск неудачный. Во втором варианте добавляется третий параметр, задающий бинарный предикат, с помощью которого соседние элементы можно проверять не только на равенство. Ниже показан пример, где с двумя вариантами набора параметров организуется поиск в векторе первой пары совпадающих элементов и первой пары, где левый элемент больше правого:

```
//Для вектора с инициализацией {1, 5, 2, 8, 1, 4, 9, 6, 6}:  
auto i = std::adjacent_find(vec_1st.begin(),  
                             vec_1st.end());  
std::cout << *i << ' ' << *(i + 1) << '\n'; //Вывод: 6 6  
i = std::adjacent_find(vec_1st.begin(),  
                             vec_1st.end(),  
                             std::greater<int>());  
std::cout << *i << ' ' << *(i + 1) << '\n'; //Вывод: 5 2
```

- *count()*, *count\_if()* – используются для подсчета количества элементов контейнера в диапазоне, заданном парой итераторов, равных указанному значению (*count*) или удовлетворяющих условию предиката (*count\_if*). Пример подсчета количества разных значений в составе вектора:

*//Для вектора с инициализацией {1, 5, 2, 8, 1, 4, 9, 6, 6}:*

*//Результат 2 (количество в векторе единиц):*

```
int amount = std::count(vec_1st.begin(),
                        vec_1st.end(),
                        1);
```

*//Результат 0 (количество в векторе нулей):*

```
amount = std::count(vec_1st.begin(), vec_1st.end(), 0);
```

- *find()*, *find\_if()* – используются для поиска элемента контейнера в диапазоне, заданном парой итераторов, равного указанному значению (*find*) или удовлетворяющего условию предиката (*find\_if*). В случае успешного поиска возвращается итератор, указывающий на самый левый из встреченных элементов, в случае неуспеха – указывающий на элемент, следующий за последним в контейнере. Пример поиска разных значений в составе вектора:

*//Предикат для отбора значений больших 5 и меньших 10:*

```
bool gr_5_and_ls_10(int x) {
    return x > 5 && x < 10;
}
```

...

*//Поиск первого из элементов, равного 6:*

```
auto f = std::find(vec_1st.begin(), vec_1st.end(), 6);
```

```
std::cout << *f << '\n'; //Вывод в консоль 6
```

*//Поиск первого из элементов, большего 5 и меньшего 10:*

```
f = std::find_if(vec_1st.begin(),
                vec_1st.end(),
                gr_5_and_ls_10);
std::cout << *f << '\n'; //Вывод в консоль 8
```

- *find\_first\_of()* – используются для поиска первого вхождения в первую последовательность любого элемента из второй последовательности. Границы обеих последовательностей задаются итераторами. В случае успешного поиска возвращается итератор, указывающий на найденный элемент в первой последовательности, в случае неуспеха – указывающий на элемент, следующий за последним в первой последовательности. Есть также вариант алгоритма, где имеется дополнительный параметр – бинарный предикат, сравнивающий элементы двух последовательностей. Пример поиска в составе вектора *vec\_1st* любого из элементов вектора *vec\_2nd*:

```
//Для вектора vec_1st с инициализацией {1, 5, 2, 8, 1, 4, 9, 6, 6}  
//и вектора vec_2nd с инициализацией {8, 1, 4}:  
auto ffo = std::find_first_of(vec_1st.begin(),  
                             vec_1st.end(),  
                             vec_2nd.begin(),  
                             vec_2nd.end());  
  
//Вывод в консоль найденного элемента и следующего за ним, т.е.: 1 5:  
std::cout << *ffo << ' ' << *(ffo+1) << '\n';
```

- *for\_each()* – алгоритм для каждого элемента последовательности, границы которой задаются двумя итераторами, вызывает указанную функцию (третий параметр). По сути это некий аналог рассмотренного выше цикла *for* в стиле *for-each*, который появился в языке относительно недавно. Основное отличие в том, что алгоритм *for\_each()* всегда работает в немодифицирующем варианте. Пример вывода в консоль данным алгоритмом элементов вектора *vec\_2nd* с помощью вспомогательной функции, выводящей одиночный элемент контейнера:

```

//Функция вывода в консоль одиночного элемента:
void output(int x) {
    std::cout << x << ' ';
}

...

//Вывод в консоль 8 1 4:
std::for_each(vec_2nd.begin(), vec_2nd.end(), output);

```

– *search()* – алгоритм находит первое вхождение в первую последовательность второй последовательности (целиком). Каждая из последовательностей задается двумя итераторами. Может быть второй вариант набора параметров, где к итераторам добавляется бинарный предикат, задающий нестандартное условие сравнения элементов последовательностей. В случае успешного поиска возвращается итератор, указывающий на найденный элемент в первой последовательности, в случае неуспеха – указывающий на элемент, следующий за последним в первой последовательности. Пример поиска в составе вектора *vec\_1st* вектора *vec\_2nd*:

```

//Для вектора vec_1st с инициализацией {1, 5, 2, 8, 1, 4, 9, 6, 6}
//и вектора vec_2nd с инициализацией {8, 1, 4}:
auto src = std::search(vec_1st.begin(),
                      vec_1st.end(),
                      vec_2nd.begin(),
                      vec_2nd.end());

//Вывод в консоль первого и последнего
//элемента найденной подпоследовательности. Результат: 8...4
std::cout << *src << "... " << *(src + 2) << '\n';

```

Далее рассмотрим некоторые из модифицирующих операций с последовательностями. В примерах в дополнение к описанным выше векторам будем использовать еще один:

```
std::vector<int> vec_tmp(3);
```

Теперь опишем следующие алгоритмы:

- *copy()* – алгоритм осуществляет копирование элементов исходной последовательности, границы которой задаются двумя итераторами, в выходную последовательность (контейнер), задаваемую итератором ее начала. Ниже показан пример копирования первых трех элементов вектора *vec\_1st* в вектор *vec\_tmp* (напоминаем, в выражении для правой границы к итератору, возвращаемому методом *begin()*, прибавляется 3, а не 2, потому что требуется указать на элемент, следующий за последним в последовательности):

*// В вектор vec\_tmp копируются элементы 1, 5, 2:*

```
std::copy(vec_1st.begin(),  
          vec_1st.begin()+3,  
          vec_tmp.begin());
```

- *fill()* – алгоритм выполняет замену элементов исходной последовательности, границы которой задаются двумя итераторами, заданным значением. Ниже показан пример замены двух последних элементов вектора *vec\_tmp* на нули:

```
std::fill(vec_tmp.begin()+1, vec_tmp.end(), 0); // 1, 0, 0
```

- *iter\_swap()*, *swap()*, *swap\_ranges()* – алгоритмы осуществляют обмен двух элементов контейнеров, заданных с помощью итераторов (*iter\_swap*) или непосредственно (*swap*). Что касается алгоритма *swap\_ranges()*, то он производит обмен элементов двух заданных последовательностей. Первая последовательность



задается двумя итераторами, вторая — одним, начальным. Последовательности должны быть одинаковой длины или емкости. Ниже показаны примеры обмена первых двух элементов *vec\_tmp* с помощью алгоритмов *iter\_swap()* и *swap()*, а также обмен и последующий повторный обмен первых трех элементов вектора *vec\_1st* и *vec\_2nd*:

```
//После обмена содержимое вектора vec_tmp: 0, 1, 0
std::iter_swap(vec_tmp.begin(), vec_tmp.begin() + 1);
//После обмена содержимое вектора vec_tmp: 1, 0, 0
std::swap(*vec_tmp.begin(), *(vec_tmp.begin() + 1));
//Для вектора vec_1st с инициализацией {1, 5, 2, 8, 1, 4, 9, 6, 6}
//и вектора vec_2nd с инициализацией {8, 1, 4},
//после обмена новое содержимое вектора vec_2nd: 1, 5, 2
std::swap_ranges(vec_1st.begin(),
                 vec_1st.begin()+3,
                 vec_2nd.begin());
//Повторный обмен, восстановление содержимого векторов:
std::swap_ranges(vec_1st.begin(),
                 vec_1st.begin()+3,
                 vec_2nd.begin());
```

- *remove()*, *remove\_if()* — алгоритмы выполняют удаление из последовательности, заданной итераторами, элементов с указанным значением (*remove*) или по предикату (*remove\_if*). Остальные элементы последовательности сдвигаются в ее начало с сохранением порядка следования. Данные алгоритмы эффективно работают только в контейнерах с произвольным доступом к элементам (вектор). В списках для этой цели имеется свой метод *remove()*, рассмотренный выше. Побочным эффектом алгоритмов является то, что размер последовательности не

меняется, «лишние» элементы в конце получают какое-то «мусорное» значение, обычно остается то, что было до сдвига. Как правило, приходится удалять такие элементы отдельным вызовом для вектора метода *erase()*. Алгоритм с этой целью возвращает итератор, указывающий на первый из этих «лишних» элементов в конце последовательности, который и используется при вызове метода *erase()*. Ниже показан пример удаления из вектора единиц с его последующим усечением:

```
//Копируем в вектор vec_tmp элементы vec_1st (1, 5, 2, 8, 1, 4, 9, 6, 6):  
vec_tmp = vec_1st;  
//Удаляем из vec_tmp единицы, усекаем его, получаем: 5, 2, 8, 4, 9, 6, 6  
vec_tmp.erase(std::remove(vec_tmp.begin(),  
                           vec_tmp.end(),  
                           1),  
               vec_tmp.end());
```

- *replace()*, *replace\_if()* – алгоритмы выполняют замену в последовательности, заданной итераторами, элементов с указанным значением (*replace*) или по предикату (*replace\_if*) на новое значение. Пример замены в векторе *vec\_tmp* с начальным содержимым (5, 2, 8, 4, 9, 6, 6) элементов со значением 6 на 3:

```
//Получаем новое содержимое вектора: 5, 2, 8, 4, 9, 3, 3:  
std::replace(vec_tmp.begin(), vec_tmp.end(), 6, 3);
```

- *reverse()* – алгоритм, изменяющий порядок следования элементов заданной последовательности на обратный. Границы последовательности, как обычно, задаются итераторами. Для списков лучше использовать одноименный метод, рассмотренный выше. Пример использования алгоритма показан ниже:

//Копируем в вектор *vec\_tmp* элементы *vec\_1st* (1, 5, 2, 8, 1, 4, 9, 6, 6):

```
vec_tmp = vec_1st;
```

//Новый порядок следования элементов вектора (6, 6, 9, 4, 1, 8, 2, 5, 1):

```
std::reverse(vec_tmp.begin(), vec_tmp.end());
```

- *unique()* – алгоритм осуществляет удаление подпоследовательности подряд идущих одинаковых элементов, оставляя только первый из них. Границы анализируемой последовательности задаются итераторами. Другой вариант алгоритма – сравнение соседних элементов с помощью бинарного предиката (в сигнатуру добавляется третий параметр). Алгоритм похож на рассмотренный ранее *remove()*, так как не изменяет размер последовательности, а возвращает итератор на новый конец данных, который можно использовать совместно с методом векторов *erase()*. Что касается списков, то для них лучше использовать одноименный метод, который был рассмотрен ранее. Пример удаления в векторе *vec\_tmp* с начальным содержимым (5, 2, 8, 4, 9, 3, 3) повторяющегося элемента со значением 3 с последующим усечением методом *erase()*:

```
vec_tmp.erase(std::unique(vec_tmp.begin(),
```

```
vec_tmp.end()),
```

```
vec_tmp.end()); //Получаем: 5, 2, 8, 4, 9, 3
```

В группе операций, связанных с сортировкой последовательностей, рассмотрим следующие алгоритмы:

- *max\_element()*, *min\_element()* – алгоритмы возвращают итераторы на наибольший и наименьший элементы последовательности соответственно. Границы последовательности задаются итераторами. Если для сравнения элементов последовательности не подходят стандартные операции «>» и «<» и/или тип

элементов контейнеров составной, то добавляется третий параметр – функциональный объект, выступающий в качестве бинарного предиката. Пример поиска в векторе *vec\_tmp* наибольшего и наименьшего элементов показан ниже:

```
//Копируем в вектор vec_tmp элементы vec_1st (1, 5, 2, 8, 1, 4, 9, 6, 6):  
vec_tmp = vec_1st;
```

```
//Вывод в консоль: max=9
```

```
std::cout << "max=" <<
```

```
    *std::max_element(vec_tmp.begin(), vec_tmp.end());
```

```
//Вывод в консоль: min=1
```

```
std::cout << "min=" <<
```

```
    *std::min_element(vec_tmp.begin(), vec_tmp.end());
```

- *stable\_partition()* – алгоритм, выполняющий перестановку элементов, удовлетворяющих заданному предикату, в начало последовательности (как обычно задается двумя итераторами). При этом порядок следования элементов сохраняется. Для примера рассмотрим перемещение в начало вектора элементов больших 5 и меньших 10. С этой целью в вызове функции в качестве третьего параметра укажем предикат *gr\_5\_and\_ls\_10()*, который определялся ранее при рассмотрении алгоритма *find()*. Соответствующий фрагмент кода приведен ниже:

```
vec_tmp = vec_1st;
```

```
//Содержимое вектора vec_tmp преобразуется от
```

```
//набора (1, 5, 2, 8, 1, 4, 9, 6, 6) к набору (8, 9, 6, 6, 1, 5, 2, 1, 4):
```

```
std::stable_partition(vec_tmp.begin(),
```

```
                    vec_tmp.end(),
```

```
                    gr_5_and_ls_10);
```

- *sort()* – алгоритм, выполняющий сортировку элементов последовательности. Границы последовательности должны быть

заданы итераторами произвольного доступа, что означает невозможность его применения для списков (там нужно использовать одноименный метод, который был рассмотрен выше). Базовый вариант метода осуществляет сортировку в порядке возрастания. Если требуется реализация другого способа сортировки, то добавляется третий параметр – функциональный объект, играющий роль бинарного предиката. Пример сортировки элементов вектора *vec\_tmp* в порядке возрастания (неубывания) показан ниже:

```
//Получаем новый порядок элементов вектора (1, 1, 2, 4, 5, 6, 6, 8, 9):  
std::sort(vec_tmp.begin(), vec_tmp.end());
```

Теперь рассмотрим пример работы с вектором с применением некоторых из рассмотренных алгоритмов.

*Составить программу, считывающую из файла in.txt значение целочисленной переменной N, элементы целочисленной матрицы A(N×N), сохраняющую их в контейнере класса vector. В исходной матрице подсчитать количество четных элементов, затем в строках с нечетными индексами удалить отрицательные элементы, а остальные строки отсортировать в порядке убывания. Найденные значения и результирующую матрицу вывести в консоль*

```
#include <algorithm>  
#include <locale>  
#include <iostream>  
#include <iomanip>  
#include <fstream>  
#include <vector>  
using namespace std;
```

```

//Предикат проверки четности элемента вектора:
bool is_even(int x) {
    return x % 2 == 0;
}

//Предикат проверки отрицательности элемента вектора:
bool is_less_than_zero(int x) {
    return x < 0;
}

//Функция вывода в консоль элементов вектора:
void output_vec(const vector<vector<int>>& v) {
    for (auto i : v) {
        for (auto j : i)
            cout << setw(4) << j;
        cout << '\n';
    }
}

int main() {
    setlocale(LC_CTYPE, "");
    int n, temp, cnt_even = 0;
    vector<vector<int>> a;
    ifstream ifs("in.txt");
    ifs >> n; //Считывание из файла размерности матрицы
    for (int i = 0; i < n; i++)
    {
        //Добавление в вектор a целочисленного вектора (строки):
        a.push_back(vector<int>());
        for (int j = 0; j < n; j++) {
            //Считывание из файла целого числа:
            ifs >> temp;

```

```

        //Добавление элемента в текущую строку:
        a[i].push_back(temp);
    }
}

cout << "Исходная матрица:" << '\n';
//Вызов функции вывода в консоль элементов вектора (матрицы):
output_vec(a);
//Определение в каждой строке (целочисленном векторе) количества
//четных элементов, накопление их в переменной cnt_even:
for (int i = 0; i < n; i++)
    cnt_even += count_if(a[i].begin(),
                        a[i].end(),
                        is_even);
cout << "Кол-во четных эл-в = " << cnt_even << '\n';
for (int i = 0; i < n; i++)
    if (i % 2 != 0)
        //Удаление из строки отрицательных элементов
        //с последующим усечением методом erase():
        a[i].erase(remove_if(a[i].begin(),
                            a[i].end(),
                            is_less_than_zero),
                    a[i].end());
    else
        //Сортировка строки с указанием стандартного
        //функционального объекта greater:
        sort(a[i].begin(),
            a[i].end(),
            greater<int>());
cout << "Преобразованная матрица:" << '\n';
output_vec(a);

```

}

Следует обратить внимание на то, что из некоторых целочисленных векторов (строк) могут удаляться элементы, то есть матрица уже может не иметь размер  $(N \times N)$ . Тем не менее, вывод вектора организуется в виде матрицы, в каждой строке после преобразования выводится текущее представление данного целочисленного вектора.

Пример работы программы:

Исходная матрица:

```
5 -12 23 18 -6
8 15 -2 10 -1
11 -9 31 -4 16
19 24 35 14 18
3 5 7 15 1
```

Кол-во четных эл-в = 11

Преобразованная матрица:

```
23 18 5 -6 -12
8 15 10
31 16 11 -4 -9
19 24 35 14 18
15 7 5 3 1
```

Также рассмотрим пример использования контейнерного класса *list*.

*Входной текстовый файл содержит строки, в каждой из которых расположены одна или несколько лексем (слов), разделенных пробелом. Необходимо считать содержимое файла, используя для хранения список, каждую из лексем, содержащую в себе подстроку "abcd" заменить на лексему "ABCD", в начале каждой строки добавить новую лексему с количеством исходных лексем, изменить порядок следования строк на обратный, исходный и результирующий списки вывести в консоль.*



```

#include <algorithm>
#include <clocale>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <list>
#include <string>
#include <sstream> //Содержит объявления строковых потоков
using namespace std;

//Функция вывода списка в консоль:
void output_list(const list < list<string>> &lst) {
    for (auto i : lst) {
        for (auto j : i)
            cout << j << ' ';
        cout << '\n';
    }
}

//Предикат проверки наличия в лексеме подстроки "abcd":
bool is_contains_abcd(string str) {
    return str.find("abcd", 0) != -1;
}

int main() {
    setlocale(LC_CTYPE, "");
    ifstream ifs("in.txt");

    //Каждая строка файла хранится как список лексем (класса string):
    list<list<string>> lst;
    string temp_line, temp_word;

```

```

//В цикле строки считываются из файла целиком, включая пробелы,
//помещаются во входной строковый поток, откуда затем
//считываются уже по одной лексеме:
while (getline(ifs, temp_line))
{
    //Инициализация строкового потока считанной строкой:
    istream iss(temp_line);
    //Добавление в конец нового списка из элементов
    //массива string:
    lst.push_back(list<string>());
    //Получаем итератор на добавленный элемент:
    auto iter = --lst.end();
    //В цикле считываем лексемы из строкового потока и
    //добавляем их в созданный список элементов массива string:
    while ((iss>>temp_word))
        (*iter).push_back(temp_word);
}
cout << "Исходный список:" << '\n';
output_list(lst);
//Цикл обработки списка:
for (auto& i : lst) {
    //В каждом списке элементов массива string заменяем
    // лексемы, содержащие подстроку "abcd" на "ABCD":
    replace_if(i.begin(),
               i.end(),
               is_contains_abcd,
               "ABCD");
    //В начало списка элементов массива string добавляем длину
    //списка, преобразованную в переменную массива string:
    i.push_front(to_string(i.size()));
}

```

```

}
// Меняем порядок следования элементов списка на обратный:
lst.reverse();
cout << "Преобразованный список:" << '\n';
output_list(lst);
}

```

Пример работы программы:

Исходный список:

```

abcd eghgfh hjjkjk fgdf
asdffds fgjabcdhj jhjhh
vcvc ghgfhfgh dgsdfa dfgggabcd
gfdg hjh jhg
sdafg

```

fgfhg hjhj fghabcdretyer

Преобразованный список:

```

3 fghfg hjhj ABCD
1 sdafg
3 gfdg hjh jhg
4 vcvc ghgfhfgh dgsdfa ABCD
3 asdffds ABCD jhjhh
4 ABCD eghgfh hjjkjk fgdf

```

## СОДЕРЖАНИЕ РАБОТЫ

### Задание А

Выбрать алгоритм, составить его блок-схему и программу, выполняющую чтение из входного файла и обработку в соответствии с индивидуальным вариантом задания двумерного массива (квадратной матрицы) заданной размерности. Структура входного файла: в первой строке расположено значение размерности матрицы  $n$ , в последующих

строках находятся  $n \times n$  элементов матрицы. Для хранения двумерного массива в памяти использовать контейнерный класс *vector*. Предусмотреть использование в программе минимум одного из алгоритмов, из числа рассмотренных в теоретических сведениях, а также различных способов перебора элементов контейнеров (через итератор, индексы, цикл в стиле *for-each*). При необходимости допускается создание вспомогательных векторов. Исходную и преобразованную матрицы вывести в консоль.

### ВАРИАНТЫ ЗАДАНИЯ

1. Дана матрица  $A(n \times n)$ . Найти сумму элементов верхней треугольной матрицы (относительно побочной диагонали). Заменить ее значением нижнюю треугольную матрицу.
2. Дана матрица  $A(n \times n)$ . Найти количество отрицательных элементов нижней треугольной матрицы (относительно побочной диагонали). Заменить этим значением отрицательные элементы верхней треугольной матрицы.
3. Дана матрица  $A(n \times n)$ . Найти наибольшие элементы каждой строки матрицы. В каждой строке заменить нулями элементы, расположенные левее наибольшего.
4. Дана матрица  $A(n \times n)$ . Найти сумму элементов каждой строки. Строки с отрицательной суммой элементов отсортировать в порядке убывания элементов.
5. Дана целочисленная матрица  $A(n \times n)$ . В каждой строке матрицы оставить только элементы, встречающиеся по одному разу. Оставшиеся элементы строк отсортировать в порядке возрастания.

6. Дана матрица  $A(n \times n)$ . Поменять местами элементы первой и последней строк матрицы. В остальных строках изменить порядок следования элементов на обратный.
7. Дана целочисленная матрица  $A(n \times n)$ , все элементы которой различны. Найти наибольший из элементов матрицы. В строке, которая содержит наибольший элемент, заменить все остальные элементы нулями.
8. Дана целочисленная матрица  $A(n \times n)$ . В строке, содержащей наибольшее количество нечетных элементов, удалить все четные элементы.
9. Дана матрица  $A(n \times n)$ , состоящая из ненулевых элементов. Если имеются несколько строк, содержащих одинаковый набор элементов, то оставить без изменений только строку с младшим индексом, а остальные заполнить нулями.
10. Дана матрица  $A(n \times n)$ . Определить наибольший и наименьший элементы матрицы и если они находятся в разных строках, то удалить из строки с наименьшим элементом все остальные элементы.
11. Дана целочисленная матрица  $A(n \times n)$ . В каждой строке определить количество четных отрицательных элементов, удалить все такие элементы из строки, где их количество наибольшее.
12. Дана матрица ненулевых элементов  $A(n \times n)$ . Определить наименьший из элементов матрицы, если в матрице он встречается несколько раз, оставить только один экземпляр (любой), остальные заменить нулями.

13. Дана матрица ненулевых элементов  $A(n \times n)$ . Определить строки, элементы которых не упорядочены в порядке возрастания, такие строки заменить нулями.
14. Дана матрица  $A(n \times n)$ . Определить наибольший элемент матрицы, если имеется несколько его вхождений, то во всех строках, его содержащих, кроме одной (любой), заменить нулями все элементы, кроме наибольшего.
15. Дана матрица ненулевых элементов  $A(n \times n)$ . Определить любую пару строк с полностью совпадающим набором элементов и изменить порядок их следования на обратный.
16. Дана матрица  $A(n \times n)$ . Заменить нулями элементы тех строк, где есть хотя бы одна пара совпадающих элементов.
17. Дана матрица  $A(n \times n)$ . Найти количество положительных элементов в каждой из строк. В той строке, где оно минимально, удалить все отрицательные элементы.
18. Дана матрица  $A(n \times n)$ . В каждой строке переместить нулевые элементы в ее начало, а остальные – удалить. В тех строках, где нет нулей, изменить порядок следования элементов на обратный.
19. Дана матрица  $A(n \times n)$ . В каждой из строк оставить только уникальные элементы, удалив дубли. Среди оставшихся элементов определить наибольший и поменять его местами с первым элементом той строки, где он находится.
20. Дана матрица  $A(n \times n)$ . Определить наибольший из элементов отдельно в верхней и нижней треугольных матрицах (относительно главной диагонали). Сравнить найденные

- значения и в той треугольной матрице, где оно окажется меньше, все элементы заменить нулями.
21. Дана матрица  $A(n \times n)$ . Определить наименьший элемент матрицы и удалить те строки, которые его содержат.
  22. Дана матрица  $A(n \times n)$ . Переставить строки матрицы таким образом, чтобы элементы первого столбца были упорядочены по неубыванию.
  23. Дана целочисленная матрица  $A(n \times n)$ . Те строки, элементы которых образуют возрастающую последовательность, оставить без изменения, в остальных удалить нечетные элементы.
  24. Дана матрица  $A(n \times n)$ . Найти значение наибольшего элемента матрицы. В той строке, где он находится переставить с сохранением порядка положительные элементы в ее начало.
  25. Дана матрица  $A(n \times n)$ , состоящая из ненулевых элементов. Заменить нулями все отрицательные элементы, а строки, полностью состоящие из положительных элементов, отсортировать в порядке возрастания.

## Задание Б

Выбрать алгоритм, составить его блок-схему и программу, выполняющую чтение текстовых данных из входного файла, сохранение их в контейнере класса *list*, обработку в соответствии со своим вариантом задания и запись результатов в выходной файл. Если в индивидуальном варианте предлагается обрабатывать отдельно каждую лексему, то при выводе результирующего списка в выходной файл разделять их одним пробелом.

Структура входного файла задается в каждом варианте задания. Предусмотреть использование в программе минимум одного из

алгоритмов, из числа рассмотренных в теоретических сведениях, а также различных способов перебора элементов контейнеров (через итератор, цикл в стиле *for-each*). При необходимости допускается создание вспомогательных списков.

## ВАРИАНТЫ ЗАДАНИЯ

1. Даны два текстовых файла, содержащих наборы строк. Проверить, совпадают ли их множества символов. В случае расхождения дополнить первый файл строкой, содержащей по одному из отсутствующих в нем символов.
2. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Удалить из файла все однобуквенные слова и лишние пробелы между словами.
3. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Удалить из файла все слова, не являющиеся палиндромами.
4. Даны текстовый файл, содержащий четное количество строк. Осуществить в нем попарный обмен строк: 1-й и 2-й, 3 и 4 и т.д..
5. Дан текстовый файл. Строки этого файла расположить в порядке убывания их длины и удалить пять самых коротких из них.
6. Даны текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Удалить те строки, в которых встречаются хотя бы одна пара стоящих рядом одинаковых слов.
7. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке файла оставить только по одному экземпляру каждого слова и



- добавить перед ними слово с количеством их вхождений в строку изначально.
8. Даны текстовые файлы  $f1, f2, f3$ , последние два несовпадающие. Если в файле  $f1$  содержатся вхождения файла  $f2$ , то заменить их на содержимое файла  $f3$ .
  9. Даны текстовые файлы  $f1, f2, f3$ . Группы символов, разделенные пробелами, будем называть словами. В файле  $f1$  оставить только те слова, которые содержатся также в файлах  $f2$  и  $f3$ .
  10. Дан текстовый файл, состоящий из строк, содержащих более одного символа и не содержащих пробелов. Заменить его содержимое перечнем строк, каждая из которых содержит один символ, встречающийся в исходном файле и упорядоченным в порядке убывания частоты их вхождений.
  11. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Если какое-то слово состоит из двух, разделенных дефисом, то разбить его на два отдельных слова, отделенных друг от друга пробелом.
  12. Дан текстовый файл, каждая строка которого представляет собой дату в формате  $dd.mm.gg$ , не совпадающую с датами в других строках с годами от 1970 до 2023. Преобразовать файл таким образом, чтобы каждая строка содержала день, наименование месяца и год в полном формате, отделенные запятой. Например: 05.01.23 преобразуется к 5, Январь, 2023. Расположить строки в порядке убывания дат.
  13. Дан текстовый файл. Удалить все строки, имеющие длину менее пяти символов. Среди оставшихся строк те, которые состоят только лишь из цифр, перенести в начало.

14. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке слова, которые не начинаются с прописной буквы заменить на слово из пяти подряд стоящих звездочек.
15. Дан текстовый файл. Определить количество раз, которые повторяются в тексте первая и последние строки. Если каждая из них встречается более одного раза, то удалить все вхождения, кроме первого (последнего).
16. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке слова, которые начинаются и заканчиваются прописной буквой переставить в начало.
17. Дан текстовый файл. Если в файле имеются пары подряд идущих одинаковых строк, то удалить все такие строки.
18. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Если какие-то слова состоят только из цифр, то заменить их словами, состоящими из пяти дефисов.
19. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке оставить только те слова, которые по своему составу соответствуют правилам образования идентификаторов в языке C++.
20. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке слова, которые содержат более двух гласных латинских букв, переставить в начало.

21. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке удалить слова, которые содержат в себе хотя бы одну цифру.
22. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Переставить в начало каждой строки те слова, которые начинаются и заканчиваются какой-либо цифрой.
23. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. Если строка содержит четное количество слов, то произвести их попарный обмен: первое и второе, третье и четвертое и т.д. В противном случае изменить порядок следования слов на обратный.
24. Дан текстовый файл, содержащий четное количество строк. Если нижняя половина списка строк (относительно середины списка) является зеркальным отображением верхней половины, то удалить ее. В противном случае упорядочить список строк в порядке убывания их длины.
25. Дан текстовый файл. Группы символов, разделенные пробелами, будем называть словами. В каждой строке слово не длиннее пяти символов, которое можно преобразовать к целому числу заменить на лексему с удвоенным значением исходного числа. Переставить такие слова в начало строки.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое STL? На каких принципах она основана?
2. Что является основными компонентами STL?
3. В чем отличия объекта класса *string* от строки с нуль-символом в конце?

4. Какие способы создания экземпляров класса *string* вам известны?
5. Как осуществляется ввод-вывод экземпляров класса *string*?
6. Какие методы класса *string* вам известны?
7. Что собой представляет контейнер класса *vector*? Какие способы создания экземпляров класса *vector* вам известны?
8. Какие операции применимы к экземплярам класса *vector*?
9. Какие способы обращения к элементам вектора вам известны?
10. Какие методы класса *vector* вам известны?
11. Какой тип итератора можно использовать при работе с классом *vector*?
12. Для чего используется ключевое слово *auto*?
13. Что такое цикл в стиле *for-each*? Как его можно применять при работе с контейнерами?
14. Как организовать перебор контейнера в обратном порядке?
15. Чем отличается контейнерный класс *list* от класса *vector*?
16. Как можно создать экземпляр класса *list*?
17. Какие методы класса *list* вам известны?
18. Что такое алгоритм в STL?
19. Какие типы алгоритмов вам известны?
20. Что такое функциональный объект? Какие predefined в STL функциональные объекты вам известны?