

ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++: ПРАКТИЧЕСКИЙ КУРС

М. В. Огнева, Е. В. Кудрина



УМО СПО рекомендует

 **Юрайт**  
издательство  
biblio-online.ru

**М. В. Огнева, Е. В. Кудрина**

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++ ПРАКТИЧЕСКИЙ КУРС



@CODELIBRARY\_IT

УЧЕБНОЕ ПОСОБИЕ ДЛЯ ВУЗОВ

*Рекомендовано Учебно-методическим отделом высшего образования в качестве  
учебного пособия для студентов высших учебных заведений, обучающихся  
по инженерно-техническим направлениям*

**Книга доступна на образовательной платформе «Юрайт» [urait.ru](#),  
а также в мобильном приложении «Юрайт.Библиотека»**

**Москва • Юрайт • 2022**

УДК 004.42(075.8)

ББК 32.973.2я73

О-38

*Авторы:*

**Огнева Марина Валентиновна** — кандидат физико-математических наук, заведующая кафедрой информатики и программирования факультета компьютерных наук и информационных технологий Федерального государственного бюджетного образовательного учреждения высшего образования «Саратовский национальный исследовательский государственный университет имени Н. Г. Чернышевского»;

**Кудрина Елена Вячеславовна** — доцент кафедры информатики и программирования факультета компьютерных наук и информационных технологий Федерального государственного бюджетного образовательного учреждения высшего образования «Саратовский национальный исследовательский государственный университет имени Н. Г. Чернышевского».

*Рецензенты:*

**Андрейченко Д. К.** — доктор физико-математических наук, профессор, заведующий кафедрой математического обеспечения вычислительных комплексов и информационных систем на базе филиала ООО «ЭпамСистэмз» Федерального государственного бюджетного образовательного учреждения высшего образования «Саратовский национальный исследовательский государственный университет имени Н. Г. Чернышевского»;

**Кондратов Д. В.** — доктор физико-математических наук, доцент, заведующий кафедрой прикладной информатики в управлении Поволжского института управления имени П. А. Столыпина — филиала Российской академии народного хозяйства и государственной службы при Президенте Российской Федерации.

**Огнева, М. В.**

О-38

Программирование на языке C++: практический курс : учебное пособие для вузов / М. В. Огнева, Е. В. Кудрина. — Москва : Издательство Юрайт, 2022. — 335 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-05123-0

Данное учебное пособие направлено на изложение базовых основ программирования на языке C++ и на формирование навыков решения практикоориентированных задач. Пособие содержит сведения по базовым концепциям структурного и объектно-ориентированного программирования, структурам данных, организации ввода-вывода, алгоритмам обработки данных, методам сортировки и т. д. Простота изложения материала и большое количество разобранных примеров делают изучение языка C++ доступным для широкого круга читателей.

Соответствует актуальным требованиям федерального государственного образовательного стандарта высшего образования.

Для студентов высших учебных заведений, обучающихся по инженерно-техническим направлениям.

УДК 004.42(075.8)

ББК 32.973.2я73

*Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.*

ISBN 978-5-534-05123-0

© Огнева М. В., Кудрина Е. В., 2017

© ООО «Издательство Юрайт», 2022

# Оглавление

<b>Предисловие .....</b>	<b>7</b>
<b>Введение.....</b>	<b>10</b>
<b>Глава 1. Базовые элементы языка С + + .....</b>	<b>14</b>
1.1. Состав языка .....	14
1.2. Структура программы.....	15
1.3. Стандартные типы данных С + + .....	18
1.4. Константы.....	20
1.5. Переменные.....	23
1.6. Организация консольного ввода/вывода данных .....	24
1.7. Операции .....	27
1.8. Выражения и преобразование типов.....	33
1.9. Примеры простейших программ .....	35
Упражнения .....	37
<b>Глава 2. Функции в С + + .....</b>	<b>40</b>
2.1. Основные понятия.....	40
2.2. Локальные и глобальные переменные .....	42
2.3. Параметры функции .....	44
2.4. Классы памяти.....	46
2.5. Модели памяти .....	48
2.6. Примеры использования функций при решении задач.....	49
Упражнения .....	51
<b>Глава 3. Операторы С + + .....</b>	<b>54</b>
3.1. Операторы следования.....	54
3.2. Операторы ветвления .....	55
3.3. Примеры использования операторов ветвления при решении задач...	59
3.4. Операторы цикла .....	64
3.5. Примеры использования операторов цикла при решении задач.....	68
3.6. Операторы безусловного перехода.....	72
Упражнения .....	74
<b>Глава 4. Рекуррентные соотношения.....</b>	<b>80</b>
4.1. Вычисление членов рекуррентной последовательности .....	80
Упражнения .....	83
<b>Глава 5. Вычисление конечных и бесконечных сумм и произведений .....</b>	<b>85</b>
5.1. Вычисление конечных сумм и произведений .....	85
5.2. Вычисление бесконечных сумм .....	91
Упражнения .....	94

<b>Глава 6. Массивы.....</b>	<b>99</b>
6.1. Указатели .....	99
6.2. Ссылки .....	104
6.3. Одномерные массивы .....	105
6.4. Примеры использования одномерных массивов .....	111
6.5. Двумерные массивы.....	115
6.6. Примеры использования двумерных массивов .....	122
6.7. Вставка и удаление элементов в массивах .....	131
Упражнения .....	139
<b>Глава 7. Строки.....</b>	<b>144</b>
7.1. Работа со строками в виде массивов символов .....	144
7.2. Класс <i>string</i> .....	150
7.3. Взаимное преобразование объектов типа <i>string</i> и строк в стиле С .....	156
7.4. Работа с отдельными символами.....	157
7.5. Смешанный строко-числовой ввод данных.....	158
7.6. Примеры работы со строками.....	159
Упражнения .....	164
Самостоятельная работа.....	166
<b>Глава 8. Рекурсивные функции. Перегрузка функций и использование шаблонов .....</b>	<b>168</b>
8.1. Рекурсивные функции.....	168
8.2. Перегрузка функций .....	176
8.3. Функции-шаблоны .....	177
Упражнения .....	179
Самостоятельная работа.....	185
<b>Глава 9. Организация файлового ввода/вывода.....</b>	<b>186</b>
9.1. Файловые потоки.....	187
9.2. Примеры решения задач с использованием файлового ввода/вывода .....	195
Упражнения .....	199
Самостоятельная работа.....	201
<b>Глава 10. Структуры.....</b>	<b>204</b>
10.1. Общие сведения.....	204
10.2. Примеры решения задач .....	208
Упражнения .....	211
Самостоятельная работа.....	213
<b>Глава 11. Сортировки .....</b>	<b>215</b>
11.1. Метод «пузырька» .....	215
11.2. Сортировка вставками .....	217
11.3. Сортировка посредством выбора.....	219
11.4. Алгоритм сортировки Шелла .....	220
11.5. Решение практических задач с использованием сортировок.....	222
Упражнения .....	226
Самостоятельная работа.....	228

<b>Глава 12. Класс-контейнер вектор</b>	<b>229</b>
12.1. Работа с векторами .....	229
12.2. Итераторы .....	234
12.3. Алгоритмы STL .....	235
Упражнения .....	240
<b>Глава 13. Исключения</b>	<b>242</b>
13.1. Механизм обработки исключений .....	242
13.2. Применение исключений на практике .....	245
Упражнения .....	247
<b>Глава 14. Классы и объекты</b>	<b>248</b>
14.1. Основные понятия.....	248
14.2. Конструкторы.....	250
14.3. Деструкторы .....	253
14.4. Статические члены класса .....	254
14.5. Перегрузка операций .....	255
14.6. Пример простого класса .....	258
Упражнения .....	261
<b>Глава 15. Наследование</b>	<b>264</b>
15.1. Основные понятия.....	264
15.2. Наследование конструкторов .....	265
15.3. Виртуальные функции .....	266
15.4. Абстрактные классы и чисто виртуальные функции .....	269
Упражнения .....	272
<b>Глава 16. Объектно-ориентированная реализация списков</b>	<b>275</b>
16.1. Основные понятия.....	276
16.2. Стек .....	276
16.3. Решение практических задач с использованием стеков.....	281
16.4. Применение исключений и шаблонов .....	284
16.5. Очередь .....	286
16.6. Решение практических задач с использованием очереди .....	290
16.7. Однонаправленный список общего вида.....	293
16.8. Решение практических задач с использованием однонаправленных списков .....	298
16.9. Двунаправленный список .....	301
16.10. Решение практических задач с использованием двунаправленных списков.....	310
Упражнения .....	312
<b>Глава 17. Реализация списков с помощью библиотеки стандартных шаблонов</b>	<b>316</b>
17.1. Класс-контейнер <i>stack</i> .....	316
17.2. Класс-контейнер <i>queue</i> .....	317
17.3. Класс-контейнер <i>list</i> .....	318
17.4. Решение практических задач с использованием библиотеки STL .....	321
Упражнения .....	324

<b>Список литературы .....</b>	<b>326</b>
<b>Приложение 1. Работа в среде Microsoft Visual Studio.....</b>	<b>328</b>
<b>Приложение 2. Ошибки, возникающие при разработке программ.....</b>	<b>331</b>
<b>Приложение 3. Операции языка C++.....</b>	<b>333</b>
<b>Приложение 4. Математические функции .....</b>	<b>335</b>

# **Предисловие**

В настоящее время в мире компьютеров существует множество языков программирования. Одну и ту же программу можно написать на языках Basic, Pascal, C/C++, C#, Java, Python. Какой из языков лучше? Ответ на данный вопрос не так прост. Однако можно с уверенностью сказать, что в языке C++ удачно сочетаются мощь, элегантность, гибкость и выразительность структурного и объектно-ориентированного программирования, благодаря чему он уже долгое время остается одним из самых популярных языков программирования.

Есть масса учебных пособий для желающих изучить язык C++. Зачем же нужно еще одно? Чем оно отличается от других?

Во-первых, для изучения данного пособия не нужны никакие предварительные знания по программированию, его может изучать любой человек, знакомый с компьютером на уровне пользователя. А простота изложения материала и большое количество разобранных примеров делают изучение языка C++ доступным для широкого круга читателей.

Во-вторых, структура учебного пособия такова, что каждая его глава содержит:

- теоретический материал;
- примеры решения типовых задач (учебных и практико-ориентированных);
- набор упражнений, рассчитанный на группу обучающихся из 15—20 человек и предназначенный для закрепления теоретического материала.

Некоторые главы содержат задания для самостоятельной работы и задачи повышенной сложности (отмечены звездочкой), что позволяет организовать творческую работу учащихся.

Такая организация позволяет реализовать все этапы обучения: изложение нового материала, закрепление материала, контроль, диагностику учебных достижений обучающихся и организацию самостоятельной работы.

В-третьих, учебное пособие разрабатывалось с учетом федеральных государственных образовательных стандартов высшего образования по ИТ-направлениям и ИТ-специальностям. Поэтому оно может быть рекомендовано:

- преподавателям вузов для подготовки и проведения занятий по таким учебным дисциплинам, как «Информатика и программирование», «Технологии программирования», «Объектно-ориентированное программирование», «Структуры данных и алгоритмы» и т.д.;

- студентам ИТ-направлений и ИТ-специальностей начальных курсов для подготовки по соответствующим учебным дисциплинам;
- а также всем заинтересованным лицам, изучающим язык С++ самостоятельно.

Данное пособие может использоваться и при обучении программированию в средней школе в рамках учебной дисциплины «Информатика и информационно-коммуникационные технологии» (профильный уровень), а также факультативных дисциплин и кружков по программированию.

Содержание и методика изложения учебного материала были одобрены, а затем внедрены в учебный процесс в рамках преподавания таких дисциплин, как «Информатика и программирование», «Технологии программирования», «Структуры данных и алгоритмы», «Объектно-ориентированное программирование» при подготовке студентов младших курсов по различным ИТ-направлениям и ИТ-специальностям Федерального государственного бюджетного образовательного учреждения высшего образования «Саратовский национальный исследовательский государственный университет имени Н. Г. Чернышевского» (факультет компьютерных наук и информационных технологий, механико-математический и физический факультеты). Следует отметить, что студенты данного вуза стабильно занимают высокие места на региональных, всероссийских и международных олимпиадах по программированию.

В результате изучения материала пособия обучающийся будет:

**знать**

- основы структурного программирования на языке С++;
- основные положения концепции объектно-ориентированного программирования и особенности их реализации на языке С++;
- специфику организации программного ввода-вывода данных на языке С++;
- способы хранения данных в программе через использование стандартных типов и линейных структур данных;
- базовые алгоритмы обработки стандартных типов и линейных структур данных, а также области их применения;
- различные методы сортировок и способы их реализации на языке С++;

**уметь**

- использовать стандартные типы данных и стандартные классы языка С++ для решения учебных и практико-ориентированных задач;
- разрабатывать собственную иерархию классов на языке С++ для решения практико-ориентированных задач;
- применять технологию объектно-ориентированного программирования для реализации линейных структур данных;

**владеть**

- навыками использования технологии структурного и объектно-ориентированного программирования;
- навыками практического программирования на языке С++.

Авторы выражают искреннюю благодарность декану факультета компьютерных наук и информационных технологий федерального государственного бюджетного образовательного учреждения высшего образования «Саратовский национальный исследовательский государственный университет имени Н. Г. Чернышевского» (СГУ), кандидату физико-математических наук, доценту, лауреату премии Президента РФ в области образования, почетному работнику высшего образования Федоровой Антонине Гавриловне. Ваша поддержка, помощь и критические замечания сыграли большую роль не только в работе над данным учебным пособием, но и в нашем профессиональном развитии.

Также авторы благодарят всех сотрудников кафедры информатики и программирования СГУ, принимавших участие в апробации содержания и методики изложения материала учебного пособия во время проведения лекционных, практических и семинарских занятий со студентами факультета компьютерных наук и информационных технологий, механико-математического и физического факультетов СГУ.

## Введение

На каком бы языке программирования ни были написаны программы, они выполняются компьютером, поэтому в окончательном виде любая программа представляет собой набор инструкций процессора. Однако пишут программы люди, поэтому практически все языки программирования предоставляют возможность более удобной для человека записи данного набора инструкций, что облегчает написание, отладку и последующую модификацию программ.

Первые программы писались на машинном языке и представляли собой двоичную или шестнадцатеричную запись команд. Потом стали использовать так называемые мнемокоды (вместо двоичной последовательности для каждой команды стало использоваться специальное обозначение). Для этой цели был изобретен язык Ассемблер, который позволил писать более длинные программы. Ассемблер относится к языкам низкого уровня, но не потому, что программы, разработанные на данном языке, «низкого» качества, а потому, что для написания самой простой программы программисту требовалось знать команды конкретного типа процессора и напрямую обращаться к данным, размещенным в его регистрах. Для перевода в машинный код программы, написанной на языке Ассемблер, стал использоваться транслятор. Появление языка Ассемблер и транслятора к нему существенно облегчило процесс разработки программ, однако со временем проявился существенный недостаток — отсутствие переносимости программ, написанных на языке Ассемблер, на компьютеры с другой архитектурой и системой команд.

Развитие вычислительной техники вело к быстрой смене типов и моделей процессоров, поэтому стало необходимо обеспечивать не только удобство в разработке, но и аппаратную переносимость программ. Это привело к появлению языков программирования высокого уровня, первым из которых был Фортран. Его существенным отличием стало то, что программа разрабатывалась на языке, «схожем» с естественным языком, в котором появились смысловые конструкции, описывающие структуры данных различной сложности и операции над ними. Также для каждой архитектуры ЭВМ стали разрабатываться платформенно-универсальные трансляторы. Теперь, чтобы перенести программу с одной аппаратной платформы на другую, следовало «перетранслировать» исходный код программы с помощью соответствующей версии транслятора.

Появление высокоуровневых языков программирования и трансляторов к ним не только решило проблему переносимости программ, но и дало толчок к развитию структурного программирования.

Первым шагом в развитии структурного программирования стало использование подпрограмм. Использование подпрограмм позволяет после их разработки и отладки отвлечься от деталей реализации. Для вызова подпрограммы требуется знать только ее интерфейс, который, если не используются глобальные переменные, полностью определяется заголовком подпрограммы.

Вторым шагом стала возможность создания собственных типов данных, позволяющих структурировать и группировать информацию, представлять ее в более естественном виде. Естественно, что для работы с собственными типами данных разрабатываются специальные подпрограммы.

Объединение в модули описаний собственных типов данных и подпрограмм для работы с ними стало следующим шагом в развитии структурного программирования. Создаваемые модули стали помещаться в библиотеку подпрограмм. Таким образом, во всех языках программирования, в том числе и в языке C++, появились обширные библиотеки стандартных подпрограмм. Следует отметить, что каждый модуль стандартной библиотеки реализует некоторую функциональность (возможность) языка, которая может использоваться другими программистами. Если в языке не хватает той или иной функциональности, создается очередной модуль и помещается в библиотеку подпрограмм.

Структурный подход в программировании и использование готовых библиотек позволили успешно создавать достаточно крупные проекты, а также уменьшить время разработки проектов и облегчить возможность их модификации. Однако сложность программного обеспечения (ПО) продолжала возрастать, и требовалось все более сложные средства ее преодоления. Идеи структурного программирования получили свое дальнейшее развитие в объектно-ориентированном программировании — технологии, позволяющей достичь простоты структуры и управляемости очень крупных программных систем.

Технология объектно-ориентированного программирования (ООП) использует идею объединения данных и подпрограмм (функций) для их обработки в специальные конструкции, получившие название объектов. Программные объекты отражают строение объектов реального мира, поэтому в качестве таких объектов могут выступать люди (сотрудники, студенты, читатели библиотеки, вкладчики банков), хранилища данных (словари, списки товаров, каталоги книг), типы данных (комплексные числа, точки на плоскости, время), структуры данных (матрицы, списки, деревья, графы) и многое другое. Программа, разработанная на основе технологии ООП, представляет собой совокупность объектов, которые взаимодействуют между собой посредством вызова функций друг друга. Технология ООП позволяет упростить процесс написания программ за счет использования таких важных принципов, как инкапсуляция, наследование и полиморфизм.

ООП основывается на понятии «класс». С точки зрения компилятора класс является типом данных, определяемым пользователем. Содержательно класс объединяет в себе данные и функции для их обработки, позволяя скрывать ту часть информации, которую не нужно видеть пользователю. В этом случае говорят, что данные и функции *инкапсулированы*. Инкапсуляция повышает степень абстракции программы — для использования в программе какого-то класса не требуется знаний о его реализации, достаточно знать только его интерфейс (описание). Это позволяет изменить реализацию класса, не затрагивая саму программу, при условии, что интерфейс класса останется прежним.

Конкретные величины типа данных «класс» называются экземплярами класса или объектами. Класс содержит члены-данные ( поля) и члены-функции (функции класса). Например, мы можем определить класс «сотрудник», в котором описать такие поля, как «фамилия», «имя», «отчество», «дата рождения», «оклад», «стаж» и многое другое, а также такие функции как «показать данные о сотруднике», «рассчитать зарплату», «рассчитать отпускные», «начислить премию» и т.д. Объекты класса «сотрудник» — это реальные люди, например, Иванов Иван Иванович, Алексеева Татьяна Сергеевна, Данилов Павел Александрович. У всех объектов значения полей данных будут различными, а вот функции по их обработке — одинаковыми, и это естественно, так как существует единое правило (алгоритм) расчета заработной платы, отпускных и премиальных. Если изменится, например, правило расчета заработной платы, то достаточно будет изменить соответствующую функцию в классе «сотрудник», и это изменение отразится на всех объектах.

В повседневной жизни мы часто сталкиваемся с разбиением классов на подклассы. Например, класс «наземный транспорт» содержит такие подклассы, как «легковые автомобили», «грузовые автомобили», «общественный транспорт» и т.д. Каждый подкласс обладает свойствами того класса, из которого он выделен. Кроме этого, каждый подкласс обладает и собственными свойствами. Так, например, и легковые, и грузовые автомобили обладают колесами и мотором и способны передвигаться по земле. Это свойства, присущие классу «наземный транспорт». В то же время подкласс «легковые автомобили» содержит небольшое количество посадочных мест для перевозки пассажиров, а подкласс «грузовые автомобили» — кузов для перевозки грузов.

Деление классов на подклассы лежит в основе принципа *наследования*. Наследование в ООП — это возможность создания иерархии классов, когда потомки (производные классы) наследуют все свойства своих предков (базовых классов), могут их изменять и добавлять новые. При этом свойства повторно не описываются, что сокращает объем программы. Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные — ближе к листьям.

Еще один важный принцип, заложенный в основу ООП, — это *полиморфизм*. Полиморфизм — это возможность использовать в рам-

ках одного класса или различных классов одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы. Частным случаем полиморфизма является перегрузка функций.

Перечислим основные преимущества ООП.

**Логичность.** Человек мыслит не терминами команд, переменных и операций, а терминами объектов, их характеристик и возможностей. С этой точки зрения ООП идеально соответствует образу человеческого мышления.

**Повторное использование кода.** Программный код, написанный в концепциях ООП, достаточно хорошо подходит для повторного использования (как непосредственного, так и с некоторыми переработками).

**Высокая скорость разработки.** Данный показатель обуславливается как упрощением процесса написания кода, так и возможностью повторного использования ранее написанного кода. В идеале, при наличии соответствующих библиотек, приложение не пишется, а собирается из готовых кусков с написанием незначительных по объему «связок».

Вместе с тем у ООП есть и недостатки:

— большие, по сравнению с остальными технологиями программирования, временные затраты на проектирование. Это тормозит начальную фазу разработки, но окупается на последующих этапах разработки ПО;

— снижение скорости работы приложения. Поддержка технологии ООП требует от среды исполнения некоторых затрат, как по времени работы, так и по объему занимаемой оперативной памяти.

Следует отметить, что развитие технологий программирования не остановилось на ООП. С развитием Windows-технологий связано появление компонентного подхода в программировании, с развитием Internet-технологий — появление технологий Web-программирования. Одним из современных достижений в области программирования стало развитие параллельного программирования. Однако неоспоримым остается то, что именно развитие принципов ООП дало толчок к появлению этих новых технологий.

# Глава 1

## БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА С++

### 1.1. Состав языка

Алфавит языка — совокупность допустимых в языке символов. Алфавит языка С++ включает:

- прописные и строчные латинские буквы и знак подчеркивания;
- арабские цифры от 0 до 9, шестнадцатеричные цифры от A до F;
- специальные символы " { } , | ; [ ] ( ) + - / % \* . \ ' : ? < = > ! & # ~ ^ ;
- пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Из символов алфавита формируются лексемы языка: идентификаторы, ключевые (зарезервированные) слова, знаки операций, константы, разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций. В свою очередь, лексемы входят в состав выражений (выражение задает правило вычисления некоторого значения) и операторов (оператор задает законченное описание некоторого действия).

#### Замечание

---

Некоторые символы, например буквы русского алфавита, не используются в С++, но их можно включать в комментарии и символьные константы.

---

**Идентификаторы** — это имена программных объектов: констант, переменных, меток, типов, экземпляров классов, функций и полей в записи. Идентификатор может включать латинские буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например *tupate*, *myName* и *MyName* — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются. Язык С++ не налагает никаких ограничений на длину имен, однако для удобства чтения и записи кода не стоит делать их слишком длинными.

**Ключевые слова** — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора, например **class**, **catch**, **int** и т.д. Ключевые слова можно использовать только по пря-

мому назначению. С ключевыми словами и их назначением можно ознакомиться в справочной системе C++.

#### Замечание

Другие лексемы (знаки операций и константы), а также правила формирования выражений и различные виды операторов будут рассмотрены чуть позже.

## 1.2. Структура программы

Каждая программа на языке C++ состоит из одной или несколько функций. Каждая функция содержит четыре основных элемента:

- 1) тип возвращаемого значения;
- 2) имя функции;
- 3) список параметров, заключенный в круглые скобки (он может быть пустым);
- 4) тело функции (последовательность операторов), которое помещается в фигурные скобки.

При запуске программы вызывается функция с именем *main* — это главная функция программы, которая должна присутствовать в программе *обязательно*. Приведем пример простейшей функции *main()*:

```
int main()
{
    return 0;
}
```

В приведенном примере:

- 1) тип возвращаемого значения **int** — целый;
- 2) имя функции *main*;
- 3) список параметров пуст;
- 4) тело функции состоит из одного оператора **return 0**, который возвратит значение 0зывающему окружению (либо компилятору, либо операционной системе).

#### Замечание

В более старых версиях C++ для функции *main()* можно было указать тип **void**. Это означает, что данная функция ничего не возвращает. В этом случае оператор *return* не нужен. В современных версиях C++ функция *main()* всегда должна возвращать тип **int**, который сообщаетзывающему окружению о нормальном или сбояном завершении работы программы. А вот список параметров функции *main()* может быть и не пуст, хотя ограничен по количеству параметров.

Теперь рассмотрим пример простейшей программы, которая подсчитывает сумму двух целых чисел, значения которых вводит с клавиатуры пользователь, а результат выводится на экран.

## Замечание

Прежде чем вводить программу в память компьютера, следует ознакомиться с принципами работы в специализированной среде разработки Microsoft Visual Studio (см. приложение 1) и с ошибками, возникающими при разработке программ (см. приложение 2).

```
#include <iostream> // директива препроцессора
/* пример программы, подсчитывающей сумму двух целых чисел, значения которых вводятся с клавиатуры, а результат выводится на экран */
using namespace std;
int main()
{
    int a,b;                                // описание переменных
    cout << "Введите 2 целых числа" << endl; // оператор вывода
    cin >> a >> b;                        // оператор ввода
    cout << "Их сумма равна " << a + b;     // оператор вывода
    return 0;
}
```

Рассмотрим первую строку кода: `#include <iostream>` — это директива препроцессора, которая указывает компилятору на необходимость подключить перед компиляцией содержимое заголовочного файла библиотеки подпрограмм C++. Имя заголовочного файла помещено в скобки, в данном случае это *iostream*. В этом файле содержатся готовые программные средства для организации форматированного ввода-вывода. В общем случае в программе может быть несколько директив, каждая из которых должна начинаться с новой строки и располагаться вне тела функции.

## Замечания

1. Название библиотеки *iostream* происходит от сокращения следующих слов *input-output* (ввод-вывод) *stream* (поток). Поток — это последовательность символов, записываемая или читаемая с устройств ввода-вывода информации каким-либо способом. Поэтому библиотеку *iostream* еще называют библиотекой для организации потокового ввода-вывода данных.

2. В некоторых версиях компилятора языка C++ для подключения заголовочных файлов вместо команды:

```
#include < имя_заголовочного_файла >;
```

вам потребуется команда

```
#include "имя_заголовочного_файла";
```

Далее идет **комментарий**. Комментарии используются для кратких заметок об используемых переменных, алгоритме или для дополнительных разъяснений сложного фрагмента кода, т.е. помогают понять смысл программы. Комментарии игнорируются компилятором, по-

этому они могут использоваться для «скрытия» части кода от компилятора.

В языке C++ существуют два вида комментариев:

1) комментарий, содержание которого умещается на одной строке, начинается с символов «//» и заканчивается символом перехода на новую строку;

2) если содержание комментария не умещается на одной строке, то используется парный комментарий, который заключается между символами «/\* \*/». Внутри парного комментария могут располагаться любые символы, включая символ табуляции и символ новой строки. Парный комментарий не допускает вложений.

#### Замечание

В нашем примере вторая и третья строки — парный комментарий.

В четвертой строке содержится директива `using namespace std`, которая означает, что все определенные ниже имена в программе будут относиться к пространству имен `std`. *Пространством имен* называется область программы, в которой определена некоторая совокупность имен. Эти имена неизвестны за пределами данного пространства имен. В нашем случае это означает, что глобальные (т.е. определенные вне программы имена из библиотеки `iostream`) имена `cout`, `cin` и `endl` определены внутри пространства имен `std`, более того, в нем определены собственные имена `a`, `b`. Пространство имен `std` — это пространство имен, используемое стандартной библиотекой. Программист вправе определить собственное пространство имен.

#### Замечание

Каждое имя, которое встречается в программе, должно быть уникальным. В больших и сложных приложениях используются библиотеки разных производителей. В этом случае трудно избежать конфликта между используемыми в них именами. Пространства имен предоставляют простой механизм предотвращения конфликтов имен. Они создают разделы в глобальном пространстве имен, разграничивая область видимости программных объектов и уникальность их имен.

Следующая строка нашей программы представляет собой заголовок функции `main`, которая не содержит параметров и должна возвращать значение типа `int`. Тело функции помещено в фигурные скобки.

В шестой строке производится объявление двух переменных типа `int` (целый тип данных). Подробнее встроенные типы языка C++ будут рассмотрены в параграфе 1.5, переменные — в параграфе 1.7.

Седьмая строка начинается с идентификатора `cout`, который представляет собой объект C++, предназначенный для работы со стандартным потоком вывода, ориентированным на экран. Далее идет операция `<<`, которая называется операцией вставки или помеще-

ния в поток. Данная операция копирует содержимое, стоящее справа от символов <<, в объект, стоящий слева от данных символов. В одной строке может быть несколько операций помещения в поток. В нашем случае в результате выполнения команды `cout <<` на экране появляется строка «Введите два целых числа», и курсор перемещается на новую строчку. Перемещение курсора на новую строку происходит за счет использования специализированного слова `endl`, называемого манипулятором: при его записи в поток вывода происходит перевод сообщения на новую строку.

Следующая строка начинается с идентификатора `cin`. Идентификатор `cin` представляет собой объект C++, предназначенный для работы со стандартным потоком ввода, ориентированным на клавиатуру. Операция `>>` называется операцией извлечения из потока: она читает значения из объекта `cin` и сохраняет их в переменных, стоящих справа от символов `>>`. В результате выполнения данной операции с клавиатуры будут введены два значения, первое из которых сохраняется в переменной `a`, второе — в переменной `b`.

Далее идет оператор вывода, в результате выполнения которого на экране появится сообщение «Их сумма равна ...». Например, если с клавиатуры были введены числа 3 и 4, то на экране появится сообщение «Их сумма равна 7».

### Замечания

Если у вас старый компилятор, то заголовок вида `#include <iostream>` придется заменить на

```
#include <iostream.h>
```

и убрать строчку

```
using namespace std
```

Из языка С язык C++ унаследовал функции ввода-вывода `scanf()` и `printf()`, их можно использовать, подключив файл `stdio.h`. Изучите данные функции самостоятельно.

## 1.3. Стандартные типы данных C++

Данные — это формализованное (понятное для компьютера) представление информации. В программах данные фигурируют в качестве значений переменных или констант. Данные, которые не изменяются в процессе выполнения программы, называются *константами*. Данные, объявленные в программе и изменяемые в процессе ее выполнения, называются *переменными*. Особенности представления данных:

1) каждое значение (переменной, константы и результата, возвращаемого функцией) имеет свой тип;

- 2) тип переменной или константы объявляется при их описании;
- 3) тип определяет:
  - внутреннее представление данных в памяти компьютера,
  - объем оперативной памяти, необходимой для размещения значения данного типа,
  - множество значений, которые могут принимать величины этого типа,
  - операции и функции, которые можно применять к величинам этого типа.

Все типы данных можно разделить на простые и составные. К простым относятся: стандартные (целые, вещественные, символьные, логический) и определенные пользователем (перечислимые типы). К составным типам относятся массивы, строки, объединения, структуры, файлы и объекты. Кроме этого существует специальный тип **void**, который не предназначен для хранения значений и применяется обычно для определения функций, которые не возвращают значения.

В данном параграфе мы рассмотрим только стандартные типы данных.

**Целые типы данных.** Существует семейство целых типов данных — **short**, **int**, **long**; и два спецификатора — **signed** и **unsigned**.

Таблица 1.1

Тип	Диапазон значений	Размер, байт
<b>short</b>	-32 768—32 767	2
<b>unsigned short</b>	0—65 535	2
<b>int</b>	-32 768—32 767 или -2 147 483 648—2 147 483 647	2, 4 или 8
<b>unsigned int</b>	0—65 535 или 0—4 294 967 295	2, 4 или 8
<b>long</b>	-2 147 483 648—2 147 483 647	4
<b>unsigned long</b>	0—4 294 967 295	4
<b>long int</b>	- $2^{31}$ —( $2^{31} - 1$ )	8

Тип **int** является аппаратно-зависимым. Это означает, что для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного — 4 байта, для 64-разрядного процессора — 8 байт. Размер типов **short**, **long** и **long int** всегда фиксирован.

Спецификатор **unsigned** используется для установления беззнакового формата представления данных, **signed** — знакового. По умолчанию все целые типы являются знаковыми, и поэтому спецификатор **signed** можно не использовать.

### Замечание

Символьные типы **char** и **wchar\_t** также могут использоваться для хранения целых чисел, но мы не будем рассматривать эти типы в данном контексте.

**Вещественные типы данных.** Существует семейство вещественных типов данных: *float*, *double*.

Таблица 1.2

Тип	Диапазон значений	Размер, байт	Количество значащих цифр после запятой
float	$3,4e - 38$ — $3,4e + 38$	4	7
double	$1,7e - 308$ — $1,7e + 308$	8	15

### Замечание

В табл. 1.2 приведены абсолютные величины минимальных и максимальных значений.

**Символьные типы.** Для описания символьных данных используются типы *char* и *wchar\_t*.

Под величину типа **char** отводится 1 байт, что достаточно для размещения любого символа из 256-символьного набора ASCII. Под величину типа **wchar\_t** отводится 2 байта, что достаточно для размещения любого символа из кодировки Unicode.

**Логический тип.** Для описания логических данных используется тип *bool*, который может принимать всего два значения: *true* (истина) и *false* (ложь). Теоретически размер переменной типа **bool** должен составлять 1 бит, но на практике компиляторы выделяют под него 1 байт, так как организовать доступ к байту на аппаратном уровне легче, чем к биту.

## 1.4. Константы

**Константами** называются данные, которые не изменяются в процессе выполнения программы. В C++ можно использовать именованные и неименованные константы.

**Неименованные** константы или **литералы** — это обычные фиксированные значения. Например, в операторе:

```
a = b + 2.5; // 2.5 - неименованная константа
```

Различаются целые, вещественные, символьные и строковые литералы. Компилятор относит литерал к одному из типов данных по его внешнему виду.

Для записи **целочисленного** литерала можно использовать одну из трех форм: десятичную, восьмеричную или шестнадцатеричную. Десятичная форма: последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль. Восьмеричная: нуль, за которым следуют восьмеричные цифры (0, 1, 2, 3, 4, 5, 6, 7). Шестнадцатеричная: 0x или 0X, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Например, значение 20 можно записать:

- 1) в десятичной форме как 20;
- 2) в восьмеричной форме как 024;
- 3) в шестнадцатеричной форме как 0x14.

По умолчанию целочисленные литералы имеют тип `int` или `long`.

Реальный тип зависит от значения. Добавив суффикс, можно явно задать тип целочисленной константы: `long`, `unsigned`, `unsigned long`.

Например, `1L` (тип `long`), `8Lu` (`unsigned long`), `128u` (`unsigned`).

Для записи вещественного литерала можно использовать десятичную или экспоненциальную форму.

Десятичная форма записи имеет вид **[целая часть].[дробная часть]**, например `2.02`, `-10.005`. При этом в записи числа может быть опущена либо целая часть, либо дробная, но не обе сразу, например `2` или `.002`.

Экспоненциальная форма имеет вид **[мантиза]{E|e}[+|-][порядок]**. В этом случае значение константы определяется как произведение мантиссы на  $10$  в степени, определенной порядком числа. Например, запись `0.2E6` эквивалентна записи  $0.2 \cdot 10^6$ , а `1.123E-2` — эквивалентна записи  $1.123 \cdot 10^{-2}$ .

По умолчанию константы с плавающей запятой имеют тип `double`. Для указания одинарной точности после значения располагают суффикс `f` или `F`, для повышенной — суффикс `l` или `L`. Например:

Десятичная форма	Экспоненциальная форма
<code>3.14159F</code>	<code>3.14159E0f</code>
<code>.001f</code>	<code>1E-3F</code>
<code>12.345L</code>	<code>1.2345E1L</code>
<code>0</code>	<code>0e0</code>

Символьный литерал — это один или два символа, заключенные в апострофы. Например, `'f'`, `'!'`, `'3'`, `'\n'`. Большинство символьных литералов являются печатаемыми символами, т.е. они отображаются на экране в том виде, в котором записаны в апострофах. Некоторые символы являются непечатаемыми, т.е. никак не отображаются ни на экране, ни при печати. Для ввода непечатаемых символов используется управляющая последовательность. Управляющая последовательность начинается символом наклонной черты влево `«\»`. Некоторые управляющие последовательности C++ приведены в табл. 1.3.

Таблица 1.3.

Вид	Назначение	Вид	Назначение
<code>\a</code>	Звуковой сигнал, оповещение ( <code>alert</code> )	<code>\t</code>	Горизонтальная табуляция ( <code>horizontal tab</code> )
<code>\b</code>	Возврат на 1 символ ( <code>backspace</code> )	<code>\v</code>	Вертикальная табуляция ( <code>vertical tab</code> )

Окончание табл. 1.3

Вид	Назначение	Вид	Назначение
\f	Перевод страницы (formfeed)	\\"	Обратная косая черта (backslash)
\n	Перевод строки, новая строка (newline)	'	Апостроф, одинарная кавычка (single quote)
\r	Возврат каретки (carriage return)	"	Кавычки

Строковый литерал — это произвольное количество символов, помещенное в кавычки, например "Ура! Сегодня информатика!!!".

В состав строковых литералов могут входить и управляющие последовательности из табл. 1.3. Например, если внутри строки требуется записать кавычки, то их предваряют косой чертой, по которой компилятор отличает их от кавычек, ограничивающих строку: "Дж. Р. Р. Толкин\" Властелин Колец\"".

В конец каждого строкового литерала компилятором добавляется нулевой символ, представляемый управляющей последовательностью \0. Поэтому длина строки всегда на единицу больше количества символов в ее записи. Таким образом, пустая строка имеет длину 1 байт.

### Замечания

Два строковых литерала, которые расположены рядом и разделены только пробелами, символами табуляции или новой строки, объединяются в один новый строковый литерал. Это облегчает запись длинных строковых литералов, занимающих несколько отдельных строк. Существует и более примитивный способ создания длинных строк: поместив в конце строки символов символ наклонной черты влево, можно указать, что следующая строка составляет с данной строкой единое целое. Символ наклонной черты влево должен располагаться в строке последним, после него не должно быть никаких комментариев и пробелов. Кроме того, все пробелы и символы табуляции в начале следующей строки окажутся частью полученной длинной строки.

Обратите внимание на разницу между строкой из одного символа, например "A", и символьной константой 'A'. Более того, пустой строковый литерал допустим, а символьный нет.

**Именованные константы.** Если при решении задачи используются постоянные значения, имеющие смысл для этой задачи, то в программе можно определить их как именованные константы. Формат объявления именованной константы:

[<класс памяти>] const <тип> <имя именованной константы> = <выражение>;

где *класс памяти* — это спецификатор, определяющий время жизни и область видимости программного объекта (см. параграф 2.4); *выражение* определяет значение именованной константы, т.е. инициализирует ее.

Константа обязательно должна быть инициализирована при объявлении. В C++ разрешается использовать при объявлении констант выражения, операндами которых могут быть ранее определенные константы. В одном операторе можно описать несколько констант одного типа, разделяя их запятыми. Например:

```
const int i = -124;
const float k1 = 2.345, k2 = 1/k1;
```

## 1.5. Переменные

**Переменная** — это именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Во время выполнения программы значение переменной можно изменять. Перед использованием любая переменная должна быть описана. Формат описания (объявления) переменных:

```
[<класс памяти>] <тип> <имя> [ = <выражение> | (<выражение>)];
```

Например, опишем переменные *i*, *j* типа **int** и переменную *x* типа **double**:

```
int i, j;
double x;
```

Перед использованием значение любой переменной должно быть определено. Это можно сделать с помощью:

1) оператора присваивания

```
int a; // описание переменной
a = 10; // определение значения переменной
```

2) оператора ввода

```
int a; // описание переменной
cin >> a; // определение значения переменной
```

3) инициализации — определения значения переменной на этапе описания. Язык C++ поддерживает две формы инициализации переменных: инициализация копией и прямая инициализация. Например:

```
int i = 100; // инициализация копией
int i (100); // прямая инициализация
```

При одном описании можно инициализировать две и более переменных, задавая каждой собственное значение. Кроме того, в одном описании могут находиться как инициализированные, так и неинициализированные переменные. Например:

```
int i, day = 3, year = 2007;
```

Необходимо помнить о том, что при использовании неинициализированной переменной для любых целей, кроме присвоения ей значения, или использовании в операторе ввода, будет получен неопределенный результат, так как будет использовано случайное значение, находящееся в памяти по адресу, на который указывает переменная. Подобные ошибки очень трудно обнаружить. Поэтому нужно пользоваться простым правилом: если первым действием с переменной будет присвоение ей значения, то в инициализации нет необходимости, в противном случае следует обязательно присвоить ей значение при описании.

## 1.6. Организация консольного ввода-вывода данных

В параграфе 1.1 уже рассматривался ввод-вывод данных. Напомним, что в C++ для организации консольного ввода-вывода данных (ввод-вывод в режиме «черного» окна) необходимо подключить заголовочный файл *iostream*. В этом файле определены:

- 1) объект *cin*, который предназначен для ввода данных со стандартного устройства ввода — клавиатуры;
- 2) объект *cout*, который предназначен для вывода данных на стандартное устройство вывода — экран;
- 3) операция *>>*, которая используется для извлечения данных из входного потока;
- 4) операция *<<*, которая используется для помещения данных в выходной поток;
- 5) операции форматированного ввода-вывода.

Рассмотрим некоторые операции форматирования выходного потока. По умолчанию данные, помещаемые в выходной поток, формируются следующим образом:

- 1) значение типа *char* помещается в поток как единичный символ и занимает в потоке поле, размерность которого равна единице;
- 2) строка помещается в поток как последовательность символов и занимает в потоке поле, размерность которого равна длине строки;
- 3) значение целого типа помещается в поток как десятичное целое число и занимает в потоке поле, размерность которого достаточна для размещения всех цифр числа, а в случае отрицательного числа еще и для знака «минус»; положительные числа помещаются в поток без знака;
- 4) значение вещественного типа помещается в поток с точностью семь цифр после «десятичной запятой»; в зависимости от величины числа оно может быть выведено в экспоненциальной форме (если число очень маленькое или очень большое) или десятичной форме.

Для форматирования потоков используются манипуляторы. Мы уже рассматривали манипулятор *endl*, который позволяет при выводе потока на экран перенести фрагмент потока на новую строку. Теперь

рассмотрим манипуляторы, которые позволяют управлять форматом вещественных типов данных и их размещением на экране.

### Замечание

Для использования манипуляторов с аргументами требуется подключить заголовочный файл *iomanip*.

**Управление форматом вещественных типов данных.** Существуют три аспекта оформления значений с плавающей запятой, которыми можно управлять: *точность* (количество отображаемых цифр); *форма записи* (десятичная или экспоненциальная); *указание десятичной точки* для значений с плавающей запятой, являющихся целыми числами.

*Точность* задает общее количество отображаемых цифр. При отображении значения с плавающей запятой округляются до текущей точности, а не усекаются. Изменить точность можно с помощью манипулятора *setprecision*. Аргументом данного манипулятора является устанавливаемая точность.

Рассмотрим следующий пример:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double i = 12345.6789;
    cout << setprecision(3) << i << endl;
    cout << setprecision(6) << i << endl;
    cout << setprecision(9) << i << endl;
    return 0;
}
```

Результат работы программы:

```
1.23e+004
12345.7
12345.6789
```

По умолчанию *форма записи* значения с плавающей запятой зависит от его размера: если число очень большое или очень маленькое, оно будет отображено в экспоненциальном формате, в противном случае — в десятичном формате. Чтобы самостоятельно установить тот или иной формат вывода, можно использовать манипуляторы *scientific* (отображать числа с плавающей запятой в экспоненциальном формате) или *fixed* (отображать числа с плавающей запятой в десятичном формате). Рассмотрим следующий пример:

```
#include <iostream>
using namespace std;
int main()
```

```

{
    double i = 12345.6789;
    cout << scientific << i << endl;
    cout << fixed << i << endl;
    return 0;
}

```

Результат работы программы:

```

1.234568e+004
12345.678900

```

Десятичная точка по умолчанию не отображается, если дробная часть равна 0. Манипулятор *showpoint* позволяет отображать десятичную точку принудительно.

```

#include <iostream>
using namespace std;
int main()
{
    double i = 10;
    cout << i << endl;
    cout << showpoint << i << endl;
    return 0;
}

```

Результат работы программы:

```

10
10.0000

```

**Управление размещением данных на экране.** Для размещения отображаемых данных используются манипуляторы:

1) *left* — выравнивает вывод по левому краю;  
 2) *right* — выравнивает вывод по правому краю;  
 3) *internal* — контролирует размещение отрицательного значения: выравнивает знак по левому краю, а значение по правому, заполняя пространство между ними пробелами;

4) *setprecision(int w)* — устанавливает максимальное количество цифр в дробной части для вещественных чисел в формате с фиксированной точкой (манипулятор *fixed*) или общее количество значащих цифр для чисел в экспоненциальном формате (манипулятор *scientific*);  
 5) *setw(int w)* — устанавливает максимальную ширину поля вывода.

Рассмотрим примеры использования данных манипуляторов.

```

#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << "1." << setw(10) << "Ivanov" << endl;
    cout << "2." << setw(10) << left << "Ivanov" << endl;
}

```

```
    cout << "3." << setw(10) << right << "Ivanov" << endl;
    return 0;
}
```

Результат работы программы:

```
1. Ivanov
2.Ivanov
3. Ivanov
```

#### Замечание

По умолчанию выравнивание происходит по правому краю (см. первый оператор вывода).

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << "1." << setw(10) << -23.4567 << endl;
    cout << "2." << setw(10) << setprecision(3) << -23.4567 << endl;
    cout << "3." << setw(10) << internal << -23.4567 << endl;
    return 0;
}
```

Результат работы программы:

```
1. -23.4567
2.      -23.5
3.-       23.5
```

#### Замечание

Обратите внимание на то, что установки манипулятора *setprecision* распространяются и на все последующие операторы вывода.

## 1.7. Операции

Полный список *операций* C++ в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой) приведен в приложении 2. Здесь мы подробно рассмотрим только часть операций, остальные операции будут вводиться по мере необходимости.

#### Замечание

Операции можно классифицировать по количеству operandов:

- на унарные, которые воздействуют на один operand;
- бинарные — воздействуют на два операнда;
- тернарные — воздействуют на три операнда.

Некоторые символы используются для обозначения как унарных, так и бинарных операций. Например, символ \* используется как для обозначения унарной операции разадресации, так и для обозначения бинарной операции умножения. Будет ли данный символ обозначать унарную или бинарную операцию, определяется контекстом, в котором он используется.

---

**Унарные операции.** Операции увеличения и уменьшения на единицу (++ и --) называются также *инкрементом* и *декрементом* соответственно. Они имеют две формы записи — *префиксную*, когда операция записывается перед операндом, и *постфиксную* — операция записывается после операнда. Префиксная операция инкремента (декремента) увеличивает (уменьшает) свой operand и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его.

Рассмотрим эти операции на примере.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 3, y = 4;
    cout << ++x << "\t" << --y << endl;
    cout << x++ << " \t" << y-- << endl;
    cout << x << "\t" << y << endl;
    return 0;
}
```

Результат работы программы:

```
4 3
4 3
5 2
```

### Замечание

Префиксная версия требует существенно меньше действий: она изменяет значение переменной и запоминает результат в ту же переменную. Постфиксная операция должна отдельно сохранить исходное значение, чтобы затем вернуть его как результат. Для сложных типов подобные дополнительные действия могут оказаться трудоемкими. Поэтому постфиксную форму имеет смысл использовать только при необходимости.

---

*Операция определения размера sizeof* предназначена для вычисления размера выражения или типа в байтах и имеет две формы: **sizeof <выражение>**, **sizeof (<тип>)**. При применении операции **sizeof** к выражению возвращается размер типа, который получается в результате вычисления выражения. При применении к типу возвращается размер заданного типа.

Рассмотрим данную операцию на примере.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 3;
    cout << sizeof (int) << endl;
    cout << sizeof (x * 10) << endl;
    cout << sizeof (x * 0.1) << endl;
    return 0;
}
```

Результат работы программы:

```
4
4
8
```

#### Замечание

Последний результат определен тем, что вещественные константы по умолчанию имеют тип *double*, к которому, как к более длинному, приводится тип всего выражения.

*Операции отрицания (-, !).* Арифметическое отрицание (или унарный минус «-») изменяет знак операнда целого или вещественного типа на противоположный.

#### Замечание

В C++ на экран в качестве значения константы **false** (ложь) выводится 0, а вместо значения константы **true** (истина) выводится 1. При этом в логическом выражении нулевое значение любого типа, в том числе и пустой указатель, преобразуется к логической константе **false**, а ненулевое значение любого типа — к логической константе **true**.

Логическое отрицание (!) дает в результате значение 0 (ложь), если operand отличен от нуля (истина), и значение 1 (истина), если operand равен нулю (ложь). Тип operandа может быть логическим, целочисленным, вещественным или указателем. Рассмотрим данные операции на примере.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 3, y = 0;
    bool f = false, v = true;
    cout << -x << "\t" << !x << endl;
    cout << -y << "\t" << !y << endl;
    cout << f << "\t" << !f << endl;
    cout << v << "\t" << !v << endl;
```

```
    return 0;  
}
```

Результат работы программы:

```
-3 0  
0 1  
0 1  
1 0
```

**Бинарные операции.** Арифметические операции — операции, применимые ко всем арифметическим типам (например к целым и вещественным типам), а также к любым другим типам, которые могут быть преобразованы в арифметические (например символьным типам). К бинарным арифметическим операциям относятся (в порядке убывания приоритета):

- 1) умножение (\*), деление (/), остаток от деления (%);
- 2) сложение (+) и вычитание (-).

Операция *деления* применима к операндам арифметического типа. Однако, если оба операнда целочисленные, то тип результата преобразуется к целому, и в качестве ответа возвращается целая часть результата деления, в противном случае тип результата определяется правилами преобразования. Операция *остаток от деления* применяется только к целочисленным операндам. Знак результата зависит от реализации.

Рассмотрим на примере операции *деления* и *остаток от деления*.

```
#include <iostream>  
using namespace std;  
int main()  
{  
    cout << 100/24 << "\t" << 100/24.0 << endl;  
    cout << 100/21 << "\t" << 100.0/24 << endl;  
    cout << 21%3 << "\t" << 21%6 << "\t" << -21%8 << endl;  
    return 0;  
}
```

Результат работы программы:

```
4   4.7619  
4   4.1667  
0   3   -5
```

**Операции отношения.** Операции отношения <, <=, >, >=, ==, != (меньше, меньше или равно, больше, больше или равно, равно, не равно соответственно) — это стандартные операции, которые сравнивают первый operand со вторым. Operandы могут быть арифметического типа или указателями. Результатом операции является значение **true** (истина) или **false** (ложь). Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

**Логические операции И и ИЛИ** (&& и ||). Логическая операция И (&&) возвращает значение **true** тогда и только тогда, когда оба операнда

принимают значение **true**, в противном случае операция возвращает значение **false**. Логическая операция ИЛИ (**||**) возвращает значение **true** тогда и только тогда, когда хотя бы один операнд принимает значение **true**, в противном случае операция возвращает значение **false**. Логические операции выполняются слева направо, при этом приоритет операции (**&&**) выше приоритета операции (**||**). Если значения первого операнда достаточно, чтобы определить результат всей операции, то второй операнд не вычисляется. Рассмотрим на примере данные операции.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "x \t y \t && \t || \t" << endl;
    cout << "0 \t 0 \t " << (0 && 0) << " \t " << (0 || 0) << endl;
    cout << "0 \t 1 \t " << (0 && 1) << " \t " << (0 || 1) << endl;
    cout << "1 \t 0 \t " << (1 && 0) << " \t " << (1 || 0) << endl;
    cout << "1 \t 1 \t " << (1 && 1) << " \t " << (1 || 1) << endl;
    return 0;
}
```

Результат работы программы:

x	y	&&	
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

### Замечание

Фактически была построена таблица истинности для логических операций И и ИЛИ.

Мы рассматриваем операции в порядке убывания их приоритета. Сейчас этот порядок будет нарушен, так как следующей по этому признаку должна быть условная операция, которая является тернарной, а мы еще не закончили рассматривать бинарные операции. Поэтому мы сначала закончим рассматривать бинарные операции присваивания, а затем перейдем к тернарной. Если вам потребуется уточнить приоритеты операций, то можно обратиться к приложению 2.

*Операции присваивания ( = , \*= , /= , %= , += , -= ). Операция простого присваивания обозначается знаком = . Формат операции простого присваивания:*

```
операнд_2 = операнд_1;
```

В результате выполнения этой операции вычисляется значение *операнда\_1*, и результат записывается в *операнд\_2*. Можно связать воедино сразу несколько операторов присваивания, записывая такие цепочки:

$a = b = c = 100$ . Присваивание такого вида выполняется справа налево: результатом выполнения  $c = 100$  является число 100, которое затем присваивается переменной  $b$ , результатом чего опять является 100, которое присваивается переменной  $a$ .

Кроме простой операции присваивания, существуют *сложные операции присваивания*, например умножение с присваиванием ( $*=$ ), деление с присваиванием ( $/=$ ), остаток от деления с присваиванием ( $%=$ ), сложение с присваиванием ( $+=$ ), вычитание с присваиванием ( $-=$ ) и т.д. (см. приложение 3).

В сложных операциях присваивания, например, при *сложении с присваиванием*, к *операнду\_2* прибавляется *операнд\_1* и результат записывается в *операнд\_2*, т.е. выражение  $c += a$  является более компактной записью выражения  $c = c + a$ . Кроме того, сложные операции присваивания позволяют генерировать более эффективный код за счет того, что в простой операции присваивания для хранения значения правого операнда создается временная переменная, а в сложных операциях присваивания значение правого операнда сразу записывается в левый операнд.

**Тернарная операция. Условная операция (? :).** Формат условной операции:

(*операнд\_1*) ? *операнд\_2* : *операнд\_3*;

*Операнд\_1* — это логическое или арифметическое выражение. Он оценивается с точки зрения его эквивалентности константам *true* (истина) и *false* (ложь). Если результат вычисления *операнда\_1* равен истине, то результатом условной операции будет значение *операнда\_2*, иначе — *операнда\_3*. Вычисляется всегда либо *операнд\_2*, либо *операнд\_3*. Их тип может различаться. Условная операция является аналогом условного оператора *if*, который будет рассмотрен позже.

Рассмотрим тернарную операцию на примере.

```
#include <iostream>
using namespace std;
int main()
{
    int x, y, max;
    cin >> x >> y;
    (x > y) ? cout << x : cout << y << endl; // 1
    max = (x > y) ? x : y;                      // 2
    cout << max << endl;
    return 0;
}
```

Результат работы программы для  $x = 11$  и  $y = 9$ :

```
11
11
```

Обратите внимание на то, что строки 1 и 2 решают одну и ту же задачу: находят наибольшее значение из двух целых чисел. Но в строке 2

в зависимости от условия ( $x > y$ ) условная операция записывает в переменную *max* либо значение *x*, либо значение *y*, после чего значение переменной *max* можно неоднократно использовать. В строке 1 наибольшее значение просто выводится на экран, и в дальнейшем это значение использовать будет нельзя, так как оно не было сохранено ни в какой переменной.

## 1.8. Выражения и преобразование типов

**Выражение** — это синтаксическая единица языка, определяющая способ вычисления некоторого значения. Выражения состоят из *операндов*, *операций* и *скобок*. Каждый operand является в свою очередь выражением или одним из его частных случаев — константой, переменной или функцией.

### Замечание

Математические функции заголовочного файла *math* (*cmath*), которые могут применяться в арифметических выражениях, приведены в приложении 4.

Примеры выражений:

```
(a + 0.12) / 6  
x && y || !z  
(t * sin(x) - 1 .05e4) / ((2 * k + 2) * (2 * k + 3))
```

Операции выполняются в соответствии с *приоритетами* (см. приложение 3). Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, то унарные операции, условная операция и операции присваивания выполняются *справа налево*, остальные — *слева направо*. Например

$a = b = c$  означает  $a = (b = c)$ ,  
 $a + b + c$  означает  $(a + b) + c$ .

Порядок вычисления подвыражений внутри выражений не определен: например, нельзя считать, что в выражении

$(\sin(x + 2) + \cos(y + 1))$

обращение к синусу будет выполнено раньше, чем к косинусу, и что  $x + 2$  будет вычислено раньше, чем  $y + 1$ .

Результат вычисления выражения характеризуется значением и типом. Например, если *a* и *b* — переменные целого типа и описаны так:

```
int a = 2, b = 5;
```

то выражение *a + b* имеет значение 7 и тип *int*.

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, то перед вычислениями выполняются преобразования более коротких типов в более длинные для сохранения значимости и точности. Иерархия типов данных приведена в табл. 1.4.

Таблица 1.4

Тип данных	Старшинство
double	Высший
float	
long	
int	
short	
char	Низший

Преобразование типов в выражениях происходит *неявно* (без участия программистов) следующим образом: если их операнды имеют различные типы, то operand с более «низким» типом автоматически будет преобразован к более «высокому» типу. Тип **bool** в арифметических выражениях преобразуется к типу **int**, при этом константа *true* заменяется единицей, а *false* — нулем.

В общем случае неявные преобразования могут происходить с потерей точности или без потери точности. Рассмотрим небольшой пример.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 100, b;
    float c = 4.5, d;
    d = a/c;           // 1 - без потери точности
    cout << "d = " << d << endl;
    b = a/c;           // 2 - с потерей точности
    cout << "b = " << b << endl;
    return 0;
}
```

Результат работы программы:

```
d = 22.2222
b = 22
```

В строке 1 переменная *a* (тип **int**) делится на переменную *c* (тип **float**). В соответствии с иерархией типов результат операции деления будет преобразован к типу **float**. Полученное значение записывается в переменную *d* (тип **float**). Таким образом, в строке 1 было выполнено одно преобразование от «низшего» типа к «высшему» и мы получили точный результат.

В строке 2 в результате операции деления также было получено значение, тип которого **float**. Но этот результат был записан в пере-

менную *b* (тип **int**). Поэтому произошло еще одно преобразование — от «высшего» типа к «низшему», что повлекло потерю точности.

Рассмотренные преобразования происходили неявно (автоматически), т.е. без участия программиста. Однако программист может совершать подобные преобразования сам с помощью операций явного преобразования типов (см. приложение 3). Для демонстрации этих операций рассмотрим небольшой пример.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 1500000000;
    a = (a * 10) / 10; // 1 - неверный результат
    cout << "a = " << a << endl;
    int b = 1500000000;
    b = (static_cast<double>(b) * 10) / 10; // 2 - верный результат
    cout << "b = " << b << endl;
    return 0;
}
```

Результат работы программы:

```
a = 211509811
b = 1500000000
```

В строке 1 мы умножили переменную *a* на 10 и получили результат, равный 15 000 000 000, который нельзя сохранить даже с помощью **unsigned int**. В этом случае было выполнено неявное преобразование типов, которое привело к потере точности вычисления.

В строке 2 перед умножением переменной *b* на 10 было выполнено явное преобразование значения переменной к типу **double**. Полученное значение было сохранено во временную переменную. Затем временная переменная умножается на 10, и поскольку результат 150 000 000 000 попадает в диапазон допустимых значений для типа **double**, то переполнения не происходит. После деления на 10 тип **double** неявно преобразуется к типу **int**, и мы получаем верный результат.

#### Замечание

Явное преобразование типов следует использовать только в случае полной уверенности в его необходимости и понимания того, для чего оно делается. Так как в случае явного преобразования компилятор не может проконтролировать корректность действий при изменении типов данных, то повышается возможность ошибок.

## 1.9. Примеры простейших программ

- Составить программу, которая для заданного значения *x* вычисляет значение выражения  $\frac{x^2 + \sin(x+1)}{25}$ .

*Указание по решению задачи.* Прежде чем составлять программу, перепишем данное выражение с учетом приоритета операций по правилам языка C++:  $(\text{pow}(x, 2) + \sin(x + 1))/25$ .

```
#include <iostream>
#include <iomanip>
#include <math.h>
using namespace std;
int main()
{
    int x;
    double y;
    cin >> x;
    y = (\text{pow}(x, 2) + \sin(x + 1))/25; // 1
    cout << "y = " << setprecision(5) << y << endl;
    return 0;
}
```

Результат работы программы при  $x = 10$ :  
y = 3.96

#### Замечание

В некоторых версиях компилятора функции *pow* и *sin* могут обрабатывать только вещественные числа. В этом случае вам потребуется изменить строку 1 следующим образом:  $(\text{pow}(x, 2.0) + \sin(x + 1.0))/25$ .

2. Написать программу, подсчитывающую площадь квадрата, периметр которого равен  $p$ .

*Указания по решению задачи.* Прежде чем составить программу, проведем математические рассуждения. Пусть дан квадрат со стороной  $a$ , тогда:

$$\begin{aligned} \text{периметр вычисляется по формуле } p = 4a &\Rightarrow a = \frac{p}{4} \\ \text{площадь вычисляется по формуле } s = a^2 &\Rightarrow a = \sqrt{s} \Rightarrow s = \left(\frac{p}{4}\right)^2. \end{aligned}$$

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
    float p, s;
    cout << "Введите периметр квадрата: ";
    cin >> p;
    s = \text{pow}(p/4, 2);
    cout << "Площадь данного квадрата = " << s;
    return 0;
}
```

Результат работы программы для  $p = 20$ :

Площадь данного квадрата = 25

3. Определить, является ли целое число четным.

*Указание по решению задачи.* Напомним, что число является четным, если остаток от деления данного числа на 2 равен нулю.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << "Введите x";
    cin >> x;
    (x % 2 == 0)? cout << "четное\n": cout << "нечетное\n"; // 1
    return 0;
}
```

Результат работы программы:

x	ответ
45	нечетное
88	четное

### Замечание

В строке 1 операция `%` находит остаток от деления на 2. Если число четное, то остаток будет равен 0, а в C++ нулевое значение трактуется как ложь. Если число нечетное, то остаток будет равен 1, а в C++ ненулевое значение трактуется как истина. Поэтому в строке 1 можно обойтись без операции сравнения. В этом случае условная операция будет выглядеть следующим образом:

```
(x % 2)? cout << "четное\n": cout << "нечетное\n";
```

## Упражнения

I. Написать программу, которая вычисляет значение выражения.

### Замечания

1. Для вывода результата использовать манипуляторы для форматирования выходного потока.

2. Считать, что вводимые значения  $x$  принадлежат области определения выражения.

- |  |   |  |
|--|---|--|
| 1) $10 \sin x +  x^4 - x^5 ;$                            | 2) $e^{-x} - \cos x + \sin 2xy;$                | 3) $\sqrt{x^4 + \sqrt{ x+1 }};$            |
| 4) $\frac{\sin x + \cos y}{\operatorname{tg} x} + 0,43;$ | 5) $\frac{0,125x +  \sin x }{1,5x^2 + \cos x};$ | 6) $\frac{x+y}{x+1} - \frac{xy-12}{34+x};$ |

$$\begin{array}{lll}
7) \frac{\sin x + \cos y}{\cos x - \sin y} \operatorname{tg} xy; & 8) \frac{1+e^{y-1}}{1+x^2 |y - \operatorname{tg} x|}; & 9) |x^3 - x^2| - \frac{7x}{x^3 - 15x}; \\
10) 1 + \frac{x}{3} + |x| + \frac{x^3 + 4}{2}; & 11) \frac{\ln |\cos x|}{\ln(1+x^2)}; & 12) \frac{1+\sin\sqrt{x+1}}{\cos(12y-4)}; \\
13) \frac{a^2+b^2}{1-\frac{a^3-b}{3}}; & 14) \frac{1+\sin^2(x+y)}{2+\left|x-\frac{2x}{1+x^2y^2}\right|} + x; & 15) x \cdot \ln x + \frac{y}{\cos x - \frac{x}{3}}; \\
16) \sin\sqrt{x+1} - \sin\sqrt{x-1}; & 17) \frac{\cos x}{\pi-2x} + 16x \cdot \cos xy; & \\
18) 2\operatorname{ctg} 3x - \frac{1}{12x^2+7x-5}; & 19) \frac{b+\sqrt{b^2+4ac}}{2a} - a^3 + b^{-2}; & \\
20) \ln \left| (y - \sqrt{|x|}) \left( x - \frac{y}{x + \frac{x^2}{4}} \right) \right|. & &
\end{array}$$

**II.** Написать программу, которая подсчитывает:

- 1) периметр квадрата, площадь которого равна  $a$ ;
- 2) площадь равностороннего треугольника, периметр которого равен  $p$ ;
- 3) расстояние между точками с координатами  $a, b$  и  $c, d$ ;
- 4) среднее арифметическое кубов двух данных чисел;
- 5) среднее геометрическое модулей двух данных чисел;
- 6) гипотенузу прямоугольного треугольника по двум данным катетам  $a, b$ ;
- 7) площадь прямоугольного треугольника по двум катетам  $a, b$ ;
- 8) периметр прямоугольного треугольника по двум катетам  $a, b$ ;
- 9) ребро куба, площадь полной поверхности которого равна  $s$ ;
- 10) ребро куба, объем которого равен  $v$ ;
- 11) периметр треугольника, заданного координатами вершин  $x1, y1, x2, y2, x3, y3$ ;
- 12) площадь треугольника, заданного координатами вершин  $x1, y1, x2, y2, x3, y3$ ;
- 13) радиус окружности, длина которой равна  $l$ ;
- 14) радиус окружности, площадь круга которой равна  $s$ ;
- 15) площадь равнобедренной трапеции с основаниями  $a$  и  $b$  и углом  $\alpha$  при большем основании;
- 16) площадь кольца с внутренним радиусом  $r1$  и внешним  $r2$ ;
- 17) радиус окружности, вписанной в равносторонний треугольник со стороной  $a$ ;
- 18) радиус окружности, описанной около равностороннего треугольника со стороной  $a$ ;
- 19) сумму членов арифметической прогрессии, если известны ее первый член, разность и число членов прогрессии;
- 20) сумму членов геометрической прогрессии, если известны ее первый член, знаменатель и число членов прогрессии.

**III.** Написать программу, которая определяет:

- 1) максимальное значение для двух различных вещественных чисел;
- 2) является ли заданное целое число четным;
- 3) является ли заданное целое число нечетным;

- 4) если целое число  $M$  делится на целое число  $N$ , то на экран выводится частное от деления, в противном случае выводится сообщение « $M$  на  $N$  нацело не делится»;
- 5) оканчивается ли данное целое число цифрой 7;
- 6) имеет ли уравнение  $ax^2 + bx + c = 0$  решение, где  $a, b, c$  — данные вещественные числа;
- 7) какая из цифр двухзначного числа больше: первая или вторая;
- 8) одинаковы ли цифры данного двухзначного числа;
- 9) является ли сумма цифр двухзначного числа четной;
- 10) является ли сумма цифр двухзначного числа нечетной;
- 11) кратна ли трем сумма цифр двухзначного числа;
- 12) кратна ли числу  $A$  сумма цифр двухзначного числа;
- 13) какая из цифр трехзначного числа больше: первая или последняя;
- 14) какая из цифр трехзначного числа больше: первая или вторая;
- 15) какая из цифр трехзначного числа больше: вторая или последняя;
- 16) все ли цифры трехзначного числа одинаковые;
- 17) существует ли треугольник с длинами сторон  $a, b, c$ ;
- 18) является ли треугольник с длинами сторон  $a, b, c$  прямоугольным;
- 19) является ли треугольник с длинами сторон  $a, b, c$  равнобедренным;
- 20) является ли треугольник с длинами сторон  $a, b, c$  равносторонним.

# Глава 2

## ФУНКЦИИ В С++

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является разделение ее на части — подпрограммы. Разделение программы на подпрограммы позволяет также избежать избыточности кода, поскольку подпрограммы описывают один раз, а вызывают на выполнение многократно из различных участков программы.

Как мы уже знаем, в С++ любая программа может состоять из нескольких функций, но в программе обязательно должна присутствовать функция *main* — главная функция, с которой начинается выполнение программы. Теперь мы научимся разрабатывать программы, состоящие из нескольких функций, а также рассмотрим такие важные понятия, как классы памяти и модели памяти.

### 2.1. Основные понятия

---

**Функция** — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

---

Любая функция должна быть объявлена и определена. *Объявление функции* (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, еще и *тело функции*. Функция может быть объявлена несколько раз, но определена только один раз. Синтаксис определения функции:

```
[<класс памяти>] <тип результата> <имя функции> ([<список параметров>])
{
    <тело функции>
}
```

Рассмотрим основные части определения функции.

1. *Класс памяти* (необязательный элемент описания) — это спецификатор, определяющий время жизни и область видимости программного объекта (см. параграф 2.4).

2. *Тип результата*, возвращаемого функцией, может быть любым, кроме массива или функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, то указывается тип *void*. Функция *main* должна возвращать значение типа *int*.

3. *Список параметров* определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра указываются его тип и имя. Список параметров может быть пустым.

4. *Тело функции* представляет собой последовательность описаний и операторов. Если тип результата функции не *void*, то тело функции должно содержать команду *return <возвращаемое значение>*.

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых параметров. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого значения не *void*, то она может входить в состав выражений, в частности может располагаться в правой части от оператора присваивания.

Рассмотрим пример функции, возвращающей сумму двух чисел.

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

int sum(int x, int y) // определение функции
{
    return x + y;
}

int main()           // главная функция
{
    int a = 5, b = 3, c;
    c = sum(a,b);           // 1
    cout << "sum = " << c << endl;
    cout << "sum = " << sum(a,b) << endl; // 2
    return 0;
}
```

Результат работы программы:

```
sum = 8
sum = 8
```

В данном примере функция *sum* сразу определена, поэтому ее предварительное объявление не требуется. Функция возвращает целочисленное значение, которое формируется выражением после команды *return*. На этапе определения функции были указаны два *формальных* целочисленных параметра. На этапе вызова функции (строки 1 и 2) в функцию передаются *фактические* параметры, которые по количеству и типу совпадают с формальными параметрами. Если количество фактических и формальных параметров будет различным, то компилятор

выдаст соответствующее сообщение об ошибке. Если параметры будут отличаться типами, то компилятор выполнит неявное преобразование типов. Обратите внимание на то, что в строке 1 вызов функции входит в состав выражения, располагаясь справа от знака присваивания. В общем случае выражения могут быть и более сложными. А в строке 2 результат, возвращаемый функцией, сразу помещается в выходной поток.

Рассмотрим другой пример: функция находит наибольшее значение для двух вещественных чисел.

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

float max(float x, float y); // объявление функции

int main() // главная функция
{
    float a = 5.5, b = 3.2, c = 14.1, d;
    d = max(max(a,b),c);      // 1
    cout << "max = " << d << endl;
    return 0;
}

float max(float x, float y) // определение функции
{
    return (x > y) ? x : y;
}
```

Результат работы программы:

```
max = 14.1
```

В данном примере функция вначале объявлена, а затем определена. Обратите внимание на то, что в строке 1 происходят два обращения к функции *max*. Вначале в функцию *max* будут переданы значения переменных *a* и *b* (5.5 и 3.2 соответственно). Функция вернет в качестве результата значение наибольшего из них, 5.5. А затем в функцию будут переданы значение 5.5 и значение переменной *c*, т.е. 14.1, из которых также будет найдено наибольшее значение. В результате в переменную *d* будет записано значение 14.1.

## 2.2. Локальные и глобальные переменные

Все величины, описанные внутри функции, а также ее параметры, являются **локальными** переменными. Область их видимости (см. параграф 2.4) — тело функции. При вызове функции в стеке (см. параграф 2.5) выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить

выполнение вызывающей функции. При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются.

### Замечание

Во всех примерах, рассмотренных выше, использовались только локальные переменные.

**Глобальные** переменные описываются вне функций, в том числе и вне функции *main*, поэтому они видны во всех функциях программы и могут использоваться для передачи данных между всеми функциями. Если имена глобальных и локальных переменных совпадают, то внутри функций локальные переменные «заменяют» глобальные, а после выхода из функции значение глобальной переменной восстанавливается. Однако с помощью операции доступа к области видимости (::) можно получить доступ к одноименной глобальной переменной. Рассмотрим небольшой пример.

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

int a = 100, b = 20; // глобальные переменные a и b

void f1(int a)      // локальная переменная a
{
    a += 10;          // 1
    cout << "f1:\t" << a << "\t" << ::a << endl; // 2
}

void f2 (int b)     // локальная переменная b
{
    b* = 2;           // 3
    cout << "f2:\t" << b << "\t" << a << endl; // 4
}

int main()
{
    cout << "main:\t" << a << "\t" << b << endl; // 5
    f1(a); f2(b); // 6
    cout << "main:\t" << a << "\t" << b << endl; // 7
    return 0;
}
```

Результат работы программы:

```
main: 100 20
f1: 110 100
f2: 40 100
main: 100 20
```

В строке 1 происходит изменение значения локальной переменной *a*. В строке 2 на экран выводится значение локальной перемен-

ной *a* и, через обращение к области видимости, — значение глобальной переменной *a*.

В строке 3 изменяется значение локальной переменной *b*. В строке 4 на экран выводятся значения локальных переменных *b* и *a*. Так как в функции *f2* нет локальной переменной с именем *a*, то использовать операцию `::` для обращения к глобальной переменной *a* не нужно.

В строках 5 и 7 на экран выводятся значения глобальных переменных *a* и *b*. В строке 6 происходит вызов вначале функции *f1*, затем *f2*. Так как тип результата данных функций **void**, то для вызова каждой из них достаточно указать только имя функции и список параметров.

#### Замечание

Глобальные переменные использовать не рекомендуется, поскольку они затрудняют отладку программы и препятствуют помещению функций в библиотеки общего пользования.

## 2.3. Параметры функции

Механизм **параметров** является основным способом обмена информацией между вызываемой и вызывающей функциями. Существует два способа передачи параметров в функцию: по значению и по адресу.

*При передаче параметров по значению* в стек заносятся копии значений фактических параметров, и операторы функции работают с этими копиями. Доступа к исходным значениям у параметров функций нет, и, следовательно, нет возможности их изменять. Во всех рассмотренных ранее примерах параметры передавались по значению.

*При передаче параметров по адресу* в стек заносятся копии адресов фактических параметров, функция осуществляет доступ к ячейкам памяти по этим адресам и, следовательно, может изменять исходные значения аргументов. Передача параметров по адресу делится на *передачу по указателю* и *передачу по ссылке*.

*При передаче параметра по указателю* в объявлении функции перед именем параметра указывается операция разадресации `*`, и при обращении к параметру в теле функции используется эта же операция. При вызове функции перед именем соответствующего фактического параметра указывается операция взятия адреса `&`. Например:

```
void f (int *a)
{
    (*a)++;
}

int main ()
{
    int x = 10;
```

```
f(&x);  
}
```

При передаче параметра по ссылке в объявлении функции перед именем параметра указывается операция взятия адреса &. В этом случае в теле функции и при вызове функции операция разадресации выполняется неявным образом, т.е. без участия программиста. Например:

```
void f (int &a)  
{  
    a++;  
}
```

```
int main ()  
{  
    int x = 10;  
    f(x);  
}
```

Рассмотрим на примере разницу в передаче параметров.

```
#include <iostream>  
using namespace std;  
  
void f(int a, int *b, int &c) // определение функции f  
{  
    a += 10;  
    (*b) += 10;  
    c += 10;  
    cout << "f:\t" << a << "\t" << *b << "\t" << c << endl;  
}  
  
int main()  
{  
    int x = 10, y = 20, z = 30;  
    cout << "main:\t" << x << "\t" << y << "\t" << z << endl;  
    f(x, &y, z); // вызов функции f  
    cout << "main:\t" << x << "\t" << y << "\t" << z << endl;  
    return 0;  
}
```

Результат работы программы:

```
main: 10 20 30  
f: 20 30 40  
main: 10 30 40
```

В данном примере:

- 1) параметр *a* передается по значению, поэтому фактический параметр *x* не изменил свое значение после завершения работы функции *f*;
- 2) параметр *b* передается по указателю, а *c* — по ссылке, поэтому фактические параметры *y* и *z* изменили свое значение после завершения работы функции *f*.

1. Более подробно про указатели можно прочитать в параграфе 6.1.
2. Если изменения, произошедшие с параметром внутри функции, не должны отразиться на значениях фактических параметров, то параметры передаются по значению, в противном случае — по адресу.
3. Использование ссылок вместо указателей при передаче параметров в функцию улучшает читаемость программы, избавляя ее от необходимости использовать операцию разадресации.
4. Использование передачи параметров по ссылке эффективнее передачи параметров по значению, поскольку не требует копирования параметров, что имеет значение при передаче структур данных большого объема.
5. Если требуется запретить изменение параметра, передающегося по адресу, внутри функции, используется модификатор *const*. Например:

```
void f (const int *a).
```

Внесем небольшие изменения в предыдущий пример:

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

void f(const int *b, int &c)
{
    (*b) += 10;           // 1
    c += 10;
}

int main()
{
    int y = 20, z = 30;
    f(&y, 2);           // 2
    cout << "main:\t" << y << "\t" << z << endl;
    return 0;
}
```

В строке 1 возникнет ошибка, так как мы попытались изменить константный параметр. В строке 2 также возникнет ошибка, так как константу (в нашем случае целое число 2) нельзя преобразовать в параметр, передаваемый по адресу.

## 2.4. Классы памяти

Рассмотрим основные правила использования спецификаторов класса памяти:

- необязательный спецификатор класса памяти может принимать одно из значений *auto*, *extern*, *static* и *register*;
- место описания переменной и спецификатор класса памяти определяют область действия, время жизни и область видимости переменной.

*Область действия* — это часть программы, в которой переменную можно использовать для доступа к связанной с ней области памяти.

В зависимости от области действия переменная может быть локальной или глобальной. Если переменная описана внутри блока (блок соответствует содержимому парных фигурных скобок), она называется локальной, область ее действия — от точки описания до конца текущего блока, включая все вложенные блоки. Если переменная описана вне любого блока, она называется глобальной, и областью ее действия считается файл, в котором она определена, от точки описания до его конца.

Класс памяти определяет время жизни и область видимости программного объекта (в частности, переменной). Если класс памяти не указан явным образом, он определяется компилятором исходя из контекста объявления.

Время жизни может быть постоянным (в течение выполнения программы) и времененным (в течение выполнения блока).

Областью видимости идентификатора называется часть текста программы, из которой допустим обычный доступ к связанной с идентификатором области памяти. Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он (блок) и входит в ее область действия. Однако если эта переменная глобальная, то к ней можно обратиться, используя операцию доступа к области видимости ::.

Для задания класса памяти используются следующие спецификаторы:

— **auto** — автоматическая переменная. Память под нее выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти происходит при выходе из блока, в котором описана переменная. Время ее жизни — с момента описания до конца блока. Для глобальных переменных этот спецификатор не используется, а для локальных он принимается по умолчанию, поэтому задавать его явным образом особого смысла не имеет;

— **extern** — означает, что переменная определяется в другом месте программы (в другом файле или дальше по тексту). Используется для создания переменных, доступных во всех модулях программы, в которых они объявлены. Если переменная в том же операторе инициализируется, то спецификатор **extern** игнорируется;

— **static** — статическая переменная. Время жизни — постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными. Глобальные статические переменные видны только в том модуле, в котором они описаны;

— **register** — аналогично **auto**, но память выделяется по возможности в регистрах процессора. Если такой возможности у компилятора нет, переменные обрабатываются как **auto**.

Пример описания переменных:

```
int a;           // 1 глобальная переменная a

int main()
{
    int b;       // 2 локальная переменная b
    extern int x; // 3 переменная x определена в другом месте
    static c;    // 4 локальная статическая переменная c
    a = 1;       // 5 присваивание глобальной переменной
    int a;       // 6 локальная переменная a
    a = 2;       // 7 присваивание локальной переменной
    ::a = 3;     // 8 присваивание глобальной переменной
    return 0;
}

int x = 4;      // 9 определение и инициализация x
```

В этом примере глобальная переменная *a* определена вне всех блоков. Память под нее выделяется в сегменте данных в начале работы программы, областью действия является вся программа. Область видимости — вся программа, кроме строк 6 и 7, так как в них определяется локальная переменная с тем же именем, область действия которой начинается с точки ее описания и заканчивается при выходе из блока. В строке 8 происходит обращение к глобальной переменной *a* через операцию доступа к области видимости `::`.

Переменные *b* и *c* — локальные, область их видимости — блок, но время жизни различно: память под *b* выделяется в стеке при входе в блок и освобождается при выходе из него, а переменная *c* располагается в сегменте данных и существует все время, пока работает программа.

Если при определении начальное значение переменных явным образом не задается, компилятор присваивает глобальным и статическим переменным случайное значение соответствующего типа. Автоматические переменные не инициализируются. Имя переменной должно быть уникальным в своей области действия (например, в одном блоке не может быть двух переменных с одинаковыми именами).

## 2.5. Модели памяти

Вся память, используемая программой, распределяется между сегментом данных, стеком и кучей. Дополнительно память выделяется под код самой программы.

В сегменте данных хранятся внешние (глобальные) и статические идентификаторы (имеющие спецификаторы `extern` и `static`). Память под данные идентификаторы в сегменте данных выделяется при их описании и освобождается перед непосредственным завершением работы программы.

Стек используется для хранения локальных (автоматических) идентификаторов. Память под данные идентификаторы выделяется в стеке при их описании и освобождается при завершении работы блока, в котором они описаны.

Куча используется для хранения данных, работа с которыми реализуется через указатели и ссылки. Сами указатели хранятся либо в сегменте данных, либо в стеке (в зависимости от указанного спецификатора класса памяти), а память для размещения данных динамически выделяется или освобождается в куче программистом с помощью специальных средств языка C++ (см. параграф 6.1).

## 2.6. Примеры использования функций при решении задач

1. Вычислить  $Z = (v1 + v2 + v3) / 3$ , где  $v1, v2, v3$  — объемы шаров с радиусами  $r1, r2, r3$  соответственно. Объем шара вычислять по формуле  $v = \frac{4}{3}\pi r^3$ .

*Указания по решению задачи.* При решении данной задачи эффективнее всего использовать функцию, которая будет вычислять объем шара. Радиус шара будет передаваться в качестве параметра.

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

float volume(float r) // функция вычисляет объема шара
{
    const float pi = 3.14;
    return 4.0/3 * pi * pow(r,3);
}

int main()
{
    float r1, r2, r3, z;
    cout << "Введите радиусы трех шаров = ";
    cin >> r1 >> r2 >> r3;
    z = (volume(r1) + volume(r2) + volume(r3))/3;
    cout << "z = " << z << endl;
    return 0;
}
```

Результат работы программы:

r1	r2	r3	z
1	2	3	50.24

2. Данна функция  $f(x) = x^3 - x^2 + x - 1$ . Найти значение выражения  $z = f(2a) + f(b+c)$ , где  $a, b, c$  — вещественные числа.

```
#include <iostream>
#include <cmath>
```

```

using namespace std;

float f(float x) // функция вычисляет  $f(x)$ 
{
    return pow(x,3) - pow(x,2) + x - 1;
}

int main()
{
    float a, b, c, z;
    cout << "Введите значение a, b, c = ";
    cin >> a >> b >> c;
    z = f(2 * a) + f(b + c); cout << "z = " << z << endl;
    return 0;
}

```

Результат работы программы:

a	b	c	z
1	2	3	109

3. Разработать функцию, которая увеличивает положительное число в два раза, а отрицательное число заменяет на противоположное. Продемонстрировать работу данной функции на примере.

```

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

void func(float &x)
{
    x = (x >= 0)? 2*x: -x;
}

int main()
{
    float a;
    cout << "Введите число = ";
    cin >> a;
    func(a);
    cout << "a = " << a << endl;
    return 0;
}

```

Результат работы программы:

a	ответ
12	24
0	0
-4	4

4. Данна сторона квадрата. С помощью одной функции вычислить его периметр и площадь. Определить, что у заданного квадрата больше — периметр или площадь.

```

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

void func(float x, float &p, float &s)
{
    p = 4*x;
    s = x*x;
}

int main()
{
    float a, p, s;
    cout << "a = ";
    cin >> a;
    func(a, p, s);
    (p>s)? cout << "периметр": cout << "площадь";
    return 0;
}

```

Результат работы программы:

a	ответ
1	периметр
5	площадь

## Упражнения

1. Разработать функцию  $\min(a, b)$  для нахождения минимального из двух чисел. Вычислить с ее помощью значение выражения

$$z = \min(3x, 2y) + \min(x - y, x + y).$$

2. Разработать функцию  $\min(a, b)$  для нахождения минимального из двух чисел. Вычислить с ее помощью минимальное значение из четырех чисел  $x, y, z, v$ .

3. Разработать функцию  $\max(a, b)$  для нахождения максимального из двух чисел. Вычислить с ее помощью значение выражения

$$z = \max(x, 2y - x) + \max(5x + 3y, y).$$

4. Разработать функцию  $f(x)$ , которая вычисляет значение по следующей формуле:

$$f(x) = x^3 - \sin x.$$

Определить, в какой из двух точек,  $a$  или  $b$ , функция принимает наибольшее значение.

5. Разработать функцию  $f(x)$ , которая вычисляет значение по следующей формуле:

$$f(x) = \cos(2x) + \sin(x - 3).$$

Определить, в какой из двух точек,  $a$  или  $b$ , функция принимает наименьшее значение.

6. Разработать функцию  $f(x)$ , которая возвращает младшую цифру натурального числа  $x$ . Вычислить с ее помощью значение выражения

$$z = f(a) + f(b).$$

7. Разработать функцию  $f(x)$ , которая возвращает вторую справа цифру натурального числа  $x$ . Вычислить с ее помощью значение выражения

$$z = f(a) + f(b) - f(c).$$

8. Разработать функцию  $f(n)$ , которая для заданного натурального числа  $n$  находит значение  $\sqrt{n} + n$ . Вычислить с ее помощью значение выражения

$$\frac{\sqrt{6}+6}{2} + \frac{\sqrt{13}+13}{2} + \frac{\sqrt{21}+21}{2}.$$

9. Разработать функцию  $f(n, x)$ , которая для заданного натурального числа  $n$  и вещественного  $x$  находит значение выражения  $x^n / n$ . Вычислить с помощью данной функции значение выражения

$$\frac{x^2}{2} + \frac{x^4}{4} + \frac{x^6}{6}.$$

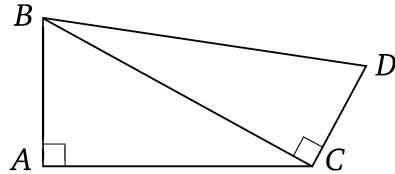
10. Разработать функцию  $f(x)$ , которая нечетное число заменяет на 0, а четное число уменьшает в два раза. Продемонстрировать работу данной функции на примере.

11. Разработать функцию  $f(x)$ , которая число, кратное пяти, уменьшает в пять раз, а остальные числа увеличивает на единицу. Продемонстрировать работу данной функции на примере.

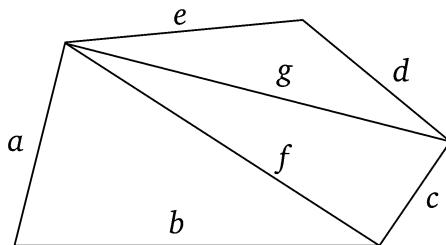
12. Разработать функцию  $f(x)$ , которая в двузначном числе меняет цифры местами, а остальные числа оставляет без изменения. Продемонстрировать работу данной функции на примере.

13. Разработать функцию  $f(x)$ , которая в трехзначном числе меняет местами первую с последней цифрой, а остальные числа оставляет без изменения. Продемонстрировать работу данной функции на примере.

14. Разработать функцию  $f(a, b)$ , которая по катетам  $a$  и  $b$  вычисляет гипотенузу. С помощью данной функции найти периметр фигуры  $ABCD$  по заданным сторонам  $AB$ ,  $AC$  и  $DC$ .



15. Разработать функцию  $f(x, y, z)$ , которая по длинам сторон треугольника  $x, y, z$  вычисляет его площадь. С помощью данной функции по заданным вещественным числам  $a, b, c, d, e, f, g$  найти площадь пятиугольника, изображенного на рисунке.



16. Разработать функцию  $f(x_1, y_1, x_2, y_2)$ , которая вычисляет длину отрезка по координатам вершин  $(x_1, y_1)$  и  $(x_2, y_2)$ , и функцию  $d(a, b, c)$ , которая вычисляет периметр треугольника по длинам сторон  $a, b, c$ . С помощью данных функций найти периметр треугольника, заданного координатами своих вершин.

17. Разработать функцию  $f(x_1, y_1, x_2, y_2)$ , которая вычисляет длину отрезка по координатам вершин  $(x_1, y_1)$  и  $(x_2, y_2)$ , и функцию  $\max(a, b)$ , которая вычисляет максимальное из чисел  $a, b$ . С помощью данных функций определить, какая из трех точек на плоскости наиболее удалена от начала координат.

18. Разработать функцию  $f(x_1, y_1, x_2, y_2)$ , которая вычисляет длину отрезка по координатам вершин  $(x_1, y_1)$  и  $(x_2, y_2)$ , и функцию  $\min(a, b)$ , которая вычисляет минимальное из чисел  $a, b$ . С помощью данных функций найти две из трех заданных точек на плоскости, расстояние между которыми минимально.

19. Разработать функцию  $f(x_1, y_1, x_2, y_2)$ , которая вычисляет длину отрезка по координатам вершин  $(x_1, y_1)$  и  $(x_2, y_2)$ , и функцию  $t(a, b, c)$ , которая проверяет, существует ли треугольник с длинами сторон  $a, b, c$ . С помощью данных функций проверить, можно ли построить треугольник по трем заданным точкам на плоскости.

20. Разработать функцию  $f(x_1, y_1, x_2, y_2)$ , которая вычисляет длину отрезка по координатам вершин  $(x_1, y_1)$  и  $(x_2, y_2)$ , и функцию  $t(a, b, c)$ , которая проверяет, существует ли треугольник с длинами сторон  $a, b, c$ . С помощью данных функций проверить, сколько различных треугольников можно построить по четырем заданным точкам на плоскости.

# Глава 3

## ОПЕРАТОРЫ С++

Программа на языке С++ состоит из последовательности **операторов**, каждый из которых определяет законченное описание некоторого действия и заканчивается точкой с запятой. Все операторы можно разделить на четыре группы: операторы следования, ветвления, цикла и операторы передачи управления.

### 3.1. Операторы следования

**Операторы следования** выполняются компилятором в естественном порядке: начиная с первого до последнего. К операторам следования относятся: *оператор-выражение* и *составной оператор*.

Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении значения выражения или выполнении законченного действия. Например:

```
++i;           // оператор инкремента
x+= y;         // оператор сложения с присваиванием
f(a, b);       // вызов функции
x = max(a, b)+a*b; // вычисление сложного выражения
```

Частным случаем оператора выражения является *пустой оператор* «;». Он используется, когда по синтаксису оператор требуется, а по смыслу — нет. В этом случае лишний символ «;» является пустым оператором и вполне допустим, хотя и не всегда безопасен. Например, случайный символ «;» после условия оператора **while** или **if** может совершенно поменять работу этого оператора.

**Составной оператор** или **блок** представляет собой последовательность операторов, заключенных в фигурные скобки. Блок обладает собственной областью видимости: объявленные внутри блока имена доступны только внутри данного блока или блоков, вложенных в него. Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Например, тело цикла **while** должно состоять только из одного оператора. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.

## 3.2. Операторы ветвления

**Операторы ветвления** позволяют изменить порядок выполнения операторов в программе. К операторам ветвлений относятся условный оператор **if** и оператор выбора **switch**.

**Условный оператор if** используется для разветвления процесса обработки данных на два направления. Он может иметь одну из форм: **сокращенную или полную**.

Формат сокращенного оператора **if**:

```
if (B)  
S;
```

где *B* — логическое или арифметическое выражение, истинность которого проверяется; *S* — один оператор: простой или составной.

При выполнении сокращенной формы оператора **if** сначала вычисляется выражение *B*, затем проводится анализ его результата: если *B* истинно, то выполняется оператор *S*; если *B* ложно, то оператор *S* пропускается. Таким образом, с помощью сокращенной формы оператора **if** можно либо выполнить оператор *S*, либо пропустить его.

Формат полного оператора **if**:

```
if (B)  
S1;  
else S2;
```

где *B* — логическое или арифметическое выражение, истинность которого проверяется; *S1, S2* — один оператор, простой или составной.

При выполнении полной формы оператора **if** сначала вычисляется выражение *B*, затем анализируется его результат: если *B* истинно, то выполняется оператор *S1*, а оператор *S2* пропускается; если *B* ложно, то выполняется оператор *S2*, а *S1* — пропускается. Таким образом, с помощью полной формы оператора **if** можно выбрать одно из альтернативных действий процесса обработки данных.

Рассмотрим несколько примеров записи условного оператора **if**.

```
if (a > 0) // Сокращенная форма с простым оператором  
x = y;  
  
if (++i) // Сокращенная форма с составным оператором  
{  
    x = y;  
    y = 2 * z;  
}  
  
if (a > 0 || b < 0) // Полная форма с простым оператором  
    x = y;  
else x = z;  
  
if (i+j-1) // Полная форма с составными операторами  
{
```

```

x = 0;
y = 1;
}
else
{
    x = 1;
    y = 0;
}

```

Операторы *S1* и *S2* могут также являться операторами **if**. Такие операторы называют вложенными. При этом ключевое слово **else** связывается с ближайшим предыдущим словом **if**, которое еще не связано ни с одним **else**. Рассмотрим несколько примеров алгоритмов с использованием вложенных условных операторов.

### Пример 1

---

```

if (A < B)
    if (C < D)          // Уровень вложенности 2
        if (E < F) X = Q; // Уровень вложенности 3
        else X = R;       // Уровень вложенности 3
    else X = Z;          // Уровень вложенности 2
else X = Y;

```

---

### Пример 2

---

```

if (A < B)          // Уровень вложенности 1
    if (C < D) X = Y; // Уровень вложенности 2
    else X = Z;       // Уровень вложенности 2
else                // Уровень вложенности 1
    if (E < F) X = R; // Уровень вложенности 2
    else X = Q;       // Уровень вложенности 2

```

---

**Оператор выбора switch** предназначен для разветвления процесса вычислений на несколько направлений. Формат оператора:

```

switch ( <выражение> )
{
    case <константное_выражение_1>: [<оператор 1>]
    case <константное_выражение_2>: [<оператор 2>]
    ...
    case <константное_выражение_n>: [<оператор n>]
    [default: <оператор> ]
}

```

Выражение, стоящее за ключевым словом **switch**, должно иметь арифметический тип или тип указатель. Все константные выражения должны иметь разные значения, но совпадать с типом выражения, стоящим после **switch**, или приводиться к нему. Ключевое слово **case** и расположенное после него константное выражение называют также меткой **case**.

Выполнение оператора начинается с вычисления выражения, расположенного за ключевым словом **switch**. Полученный результат сравни-

вается с меткой **case**. Если результат выражения соответствует метке **case**, то выполняется оператор, стоящий после этой метки. Затем последовательно выполняются все операторы до конца оператора **switch**, если только их выполнение не будет прервано с помощью оператора передачи управления **break** (см. пример). При использовании оператора **break** происходит выход из **switch**, и управление передается следующему за **switch** оператору. Если же совпадения выражения ни с одной меткой **case** не произошло, то выполняется оператор, стоящий после слова **default**, а при его отсутствии управление передается следующему за **switch** оператору.

### Пример 3

---

Известен порядковый номер дня недели. Вывести на экран его название.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    switch (x)
    {
        case 1: cout << "понедельник" ; break;
        case 2: cout << "вторник"; break;
        case 3: cout << "среда"; break;
        case 4: cout << "четверг"; break;
        case 5: cout << "пятница"; break;
        case 6: cout << "суббота"; break;
        case 7: cout << "воскресенье"; break;
        default: cout << "вы ошиблись";
    }
    return 0;
}
```

Результат работы программы:

```
x Сообщение на экране
2 вторник
4 четверг
10 вы ошиблись
```

Если исключить из этого примера операторы **break**, получим следующий результат:

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    switch (x)
    {
```

```

        case 1: cout << "понедельник" ;
        case 2: cout << "вторник";
        case 3: cout << "среда";
        case 4: cout << "четверг";
        case 5: cout << "пятница";
        case 6: cout << "суббота";
        case 7: cout << "воскресенье";
    default: cout << "вы ошиблись";
    }
    return 0;
}

```

Результат работы программы:

```

x Сообщение на экране
4 четверг пятница суббота воскресенье вы ошиблись
6 суббота воскресенье вы ошиблись
10 вы ошиблись

```

---

Существует одна стандартная ситуация, когда оператор **break** не нужен. Речь идет о случае, когда одна и та же последовательность действий должна выполняться для нескольких меток **case**. В этом случае метки **case** располагают последовательно одну за другой через двоеточие. После последней метки указывают действие, которое нужно выполнить. Например:

```

#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    switch (x)
    {
        case 1: case 2: case 3: case 4:
        case 5: cout << "рабочий день"; break;
        case 6: case 7: cout << "выходной"; break
        default: cout << "вы ошиблись";
    }
    return 0;
}

```

Результат работы программы:

```

x Сообщение на экране
4 рабочий день
6 выходной
10 вы ошиблись

```

В этом примере для меток со значениями из диапазона 1—5 выполняется одно действие, а для меток со значениями 6—7 — другое действие.

### 3.3. Примеры использования операторов ветвления при решении задач

1. Для произвольных значений аргументов вычислить значение функции, заданной следующим образом:  $y(x) = \frac{1}{x} + \sqrt{x+1}$ . Если в некоторой точке вычислить значение функции окажется невозможно, то вывести на экран сообщение «функция не определена».

*Указание по решению задачи.* Данную задачу можно решить двумя способами.

1-й способ. Заданная функция не определена в том случае, когда знаменатель первого слагаемого равен нулю,  $x = 0$ , или подкоренное выражение второго слагаемого — отрицательное число,  $x + 1 < 0$  или  $x < -1$ .

В остальных случаях функция определена. Программа выглядит следующим образом:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    float x,y;
    cout << "x : = ";
    cin >> x;
    if (!x || x < -1) // проверка условия неопределенности функции
        cout << "Функция не определена" << endl;
    else
    {
        y = 1/x+sqrt(x+1);
        cout << "x = " << x << "\t" << "y = " << y << endl;
    }
    return 0;
}
```

2-й способ. Заданная функция определена в том случае, когда знаменатель первого слагаемого не равен нулю,  $x \neq 0$ , и подкоренное выражение второго слагаемого неотрицательно,  $x + 1 \geq 0$ ,  $x \geq -1$ .

В остальных случаях функция не определена. Программа выглядит следующим образом:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    float x,y;
    cout << "x = ";
    cin >> x;
    if ((x) && (x >= -1)) // проверка условия определенности функции
    {
```

```

y = 1/x+sqrt(x+1);
cout << "x = " << x << "\t" << "y = " << y << endl;
}
else cout << "Функция не определена" << endl;
return 0;
}

```

Обе программы дадут нам следующий результат:

x	y(x)
0	функция не определена
2	1.50

**2.** Для произвольных значений аргументов вычислить значение функции, заданной следующим образом:

$$y(x) = \begin{cases} (x^3 + 1)^2, & \text{при } x < 0; \\ 0, & \text{при } 0 \leq x < 1; \\ |x^2 - 5x + 1|, & \text{при } x \geq 1. \end{cases}$$

*Указания по решению задачи.* Вся числовая прямая  $Ox$  разбивается на три непересекающихся интервала,  $(-\infty; 0)$ ,  $[0; 1)$ ,  $[1; +\infty)$ . На каждом интервале функция задается своею ветвью. Заданная точка  $x$  может попасть только в один из указанных интервалов. Чтобы определить, в какой из интервалов попала точка, воспользуемся следующим алгоритмом. Если  $x < 0$ , то  $x$  попадает в первый интервал, и функцию высчитываем по первой ветви, после чего проверка заканчивается. Если это условие ложно, то истинно условие  $x \geq 0$ , и для того чтобы точка попала во второй интервал, достаточно, чтобы выполнялось условие  $x < 1$ . Если выполняется это условие, то точка  $x$  попадает во второй интервал, и мы определяем функцию по второй ветви, после чего заканчиваем вычисления. В противном случае точка может принадлежать только третьему интервалу. Поэтому дополнительную проверку не проводим, а сразу вычисляем функцию по третьей ветви. Приведенный алгоритм можно реализовать с помощью вложенных операторов **if**.

```

#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    float x,y;
    cout << "x = ";
    cin >> x;
    if (x < 0)           // проверяем условие первой ветви
        y = pow(pow(x, 3)+1, 2);
    else if (x < 1)      // проверяем условие второй ветви
        y = 0;
    else y = fabs(x*x-5*x+1);
}

```

```

cout << "f( " << x << ") = " << y;
return 0;
}

```

Результат работы программы:

координата точки	ответ
0	0
1	3
-2	49

**3.** Данна точка на плоскости с координатами  $(x, y)$ . Составить программу, которая выдает одно из сообщений «Да», «Нет», «На границе» в зависимости от того, лежит ли точка внутри заштрихованной области, вне заштрихованной области или на ее границе.

*Указания по решению задачи.* Всю плоскость можно разбить на три непересекающихся множества точек:  $I_1$  — множество точек, лежащих внутри области;  $I_2$  — множество точек, лежащих вне области;  $I_3$  — множество точек, образующих границу области. Точка с координатами  $(x, y)$  может принадлежать только одному из них. Поэтому проверку можно проводить по аналогии с алгоритмом, приведенным в примере 2. Однако множества  $I_1$ ,  $I_2$ ,  $I_3$  значительно труднее описать математически, чем интервалы в примере 2. Поэтому для непосредственной проверки выбираются те два множества, которые наиболее просто описать математически. Обычно最难нее всего описать точки границы области. Например, для рис. 2.1 множества задаются следующим образом:

$$I_1 : x^2 + y^2 < 10^2;$$

$$I_2 : x^2 + y^2 > 10^2;$$

$$I_3 : x^2 + y^2 = 10^2.$$

Для рис. 2.2 множества задаются следующим образом:

$$I_1 : |x| < 10 \text{ и } |y| < 5;$$

$$I_2 : |x| > 10 \text{ или } |y| > 5;$$

$$I_3 : (|x| \leq 10 \text{ и } y = 5), \text{ или } (|x| \leq 10 \text{ и } y = -5),$$

$$\text{или } (|y| < 5 \text{ и } x = 10), \text{ или } (|y| < 5 \text{ и } x = -10).$$

Таким образом, для рис. 2.1 описание всех множеств равносильно по сложности, а для рис. 2.2 описать множество  $I_3$  значительно сложнее.

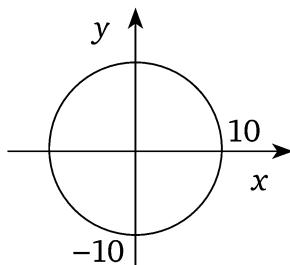


Рис. 2.1

```

#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    float x,y;
    cout << "x = ";
    cin >> x;
    cout << "y = ";
    cin >> y;
    if (x * x + y * y < 100)      // внутри области
        cout << "Да";
    else if (x * x + y * y > 100) // вне области
        cout << "Нет";
    else cout << "на границе";
    return 0;
}

```

Результаты работы программы:

координаты точек	ответ
0 0	да
10 0	на границе
-12 13	нет

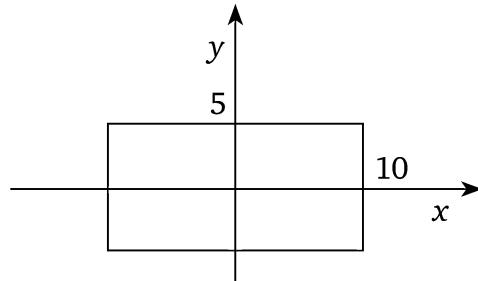


Рис. 2.2

```

#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    float x,y;
    cout << "x = ";
    cin >> x;
    cout << "y = ";
    cin >> y;
    if (fabs(x)<10 && fabs(y)<5)      // внутри области
        cout << "Да";
    else if (fabs(x)>10 || fabs(y)>5) // вне области
        cout << "Нет";
    else cout << "на границе";
    return 0;
}

```

## Результаты работы программы:

координаты точек	ответ
0 0	да
10 5	на границе
-12 13	нет

4. Дан номер фигуры (1 — квадрат, 2 — треугольник). По номеру фигуры запросить необходимые данные для вычисления площади, произвести вычисление площади фигуры и вывести полученные данные на экран.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int x;
    cout << "Программа подсчитывает площадь:\n"
        1. квадрата;\n
        2. треугольника.\n
        3. выход из программы";
    cout << "Укажите номер фигуры или завершите работу с программой.\n";
    cin >> x;
    switch (x)
    {
        case 1 :{
            cout << "введите длину стороны квадрата\n" ;
            float a;
            cin >>a;
            if (a>0)
                cout << "Площадь квадрата со стороной" << a <<
                    "равна\t" << a*a;
            else cout << "Квадрат не существует\n";
            break;
        }
        case 2: {
            cout << "введите длины сторон треугольника\n";
            float a,b,c,p, s;
            cin >> a >>b >> c;
            if (a+b>c && a+c>b && b+c>a)
            {
                p = (a+b+c)/2;
                s = sqrt(p*(p-a)*(p-b)*(p-c));
                cout << "Площадь треугольника со сторонами" << a << b << c ;
                cout << "равна\t" << s;
            }
            else cout << "Треугольник не существует\n";
            break;
        }
        case 3:break;
        default: cout << "Номер фигуры указан неверно\n";
    }
    return 0;
}
```

### 3.4. Операторы цикла

**Операторы цикла** используются для организации многократно повторяющихся вычислений. К операторам цикла относятся:

- цикл с предусловием **while**;
- цикл с постусловием **do while**;
- цикл с параметром **for**.

**Цикл с предусловием while.** Оператор цикла **while** организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Формат цикла **while**:

```
while (B)
    S;
```

где *B* — выражение, истинность которого проверяется (условие завершения цикла); *S* — тело цикла: один оператор (простой или составной).

Перед каждым выполнением тела цикла анализируется значение выражения *B*: если оно истинно, то выполняется тело цикла и управление передается на повторную проверку условия *B*; если значение *B* ложно — цикл завершается и управление передается на оператор, следующий за оператором *S*.

Если результат выражения *B* окажется ложным при первой проверке, то тело цикла не выполнится ни разу. Отметим, что если условие *B* во время работы цикла не будет изменяться, то цикл не сможет завершить работу. Возникнет ситуация зацикливания. Поэтому внутри тела должны находиться операторы, приводящие к изменению значения выражения *B* так, чтобы цикл мог корректно завершиться.

В качестве иллюстрации выполнения цикла **while** рассмотрим программу вывода на экран целых чисел из интервала от 1 до *n*.

```
#include <iostream>
using namespace std;
int main()
{
    int n, i = 1;
    cout << "n = ";
    cin >> n;
    while (i <= n)          // пока i меньше или равно n
    {
        cout << i << "\t";  // выводим на экран значение i
        ++i;                // увеличиваем i на единицу
    }
    return 0;
}
```

Результаты работы программы:

n	ответ
10	1 2 3 4 5 6 7 8 9 10

Используя операцию постфиксного инкремента, тело цикла можно заменить одной командой

```
cout << i++ << "\t".
```

**Цикл с постусловием do while.** Оператор цикла **do while** также организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Однако, в отличие от цикла **while**, условие завершения цикла проверяется после выполнения тела цикла. Формат цикла **do while**:

```
do  
    S  
while (B);
```

где *B* — выражение, истинность которого проверяется (условие завершения цикла); *S* — тело цикла: один оператор (простой или блок).

Сначала выполняется оператор *S*, а затем анализируется значение выражения *B*: если оно истинно, то управление передается оператору *S*, если ложно — цикл завершается и управление передается оператору, следующему за условием *B*.

В операторе **do while**, так же как и в операторе **while**, возможна ситуация зацикливания в случае, если условие *B* всегда будет оставаться истинным. Но так как условие *B* проверяется после выполнения тела цикла, то в любом случае тело цикла выполнится хотя бы один раз.

В качестве иллюстрации выполнения цикла **do while** рассмотрим программу вывода на экран целых чисел из интервала от 1 до *n*.

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int n, i = 1;  
    cout << "n = ";  
    cin >>n;  
    do // выводим на экран i, а затем увеличиваем  
        cout << i++ << "\t"; // ее значение на единицу  
    while (i<= n); // до тех пор пока i меньше или равна n  
    return 0;  
}
```

Результаты работы программы:

n	ответ
10	1 2 3 4 5 6 7 8 9 10

**Цикл с параметром for.** Цикл с параметром имеет следующую структуру:

```
for ( <инициализация>; <выражение>; <модификация> )  
    <оператор>;
```

*Инициализация* используется для объявления и присвоения начальных значений величинам, используемым в цикле. В этой части можно записать несколько операторов, разделенных запятой. Областью действия переменных, объявленных в части инициализации цикла, являются цикл и вложенные блоки. Инициализация выполняется один раз в начале исполнения цикла. *Выражение* определяет условие выполнения цикла: если его результат истинен, цикл выполняется. Истинность выражения проверяется перед каждым выполнением тела цикла, таким образом, цикл с параметром реализован как цикл с предусловием. *Модификация* выполняется после каждой итерации цикла и служит обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую. *Оператор* (простой или составной) представляет собой тело цикла.

Любая из частей оператора **for** (инициализация, выражение, модификация, оператор) может отсутствовать, но точку с запятой, определяющую позицию пропускаемой части, надо оставить.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++) // для i от 1 до n с шагом 1
        cout << i << "\t"; // выводить на экран значение i
    return 0;
}
```

Результаты работы программы:

n	ответ
10	1 2 3 4 5 6 7 8 9 10

#### Замечание

Используя операцию постфиксного инкремента при выводе данных на экран, цикл **for** можно преобразовать следующим образом:

```
for (int i = 1; i <= n;)
    cout << i++ << "\t";
```

В этом случае в заголовке цикла **for** отсутствует блок модификации.

**Вложенные циклы.** Циклы могут быть простыми или вложенными (кратные, циклы в цикле). Вложенными могут быть циклы любых типов: **while**, **do while**, **for**. Структура вложенных циклов на примере типа **for** приведена ниже:

```

for (i = 1; i < ik; i++)
{
    ...
    for (j = 10; j > jk; j--)
        {
            ...
            for(k = 1; k < kk; j += 2 ){...}
        }
    ...
}

```

1  
2  
3

Каждый внутренний цикл должен быть полностью вложен во все внешние циклы. «Пересечение» циклов не допускается.

Рассмотрим пример использования вложенных циклов, который позволит вывести на экран следующую таблицу:

2	2	2	2	2
2	2	2	2	2
2	2	2	2	2
2	2	2	2	2

```

#include <iostream>
using namespace std;
int main()
{
    for (int i = 1; i <= 4; ++i, cout << endl) // внешний цикл
        for (int j = 1; j <= 5; ++j) // внутренний цикл
            cout << "2\t"; // тело внутреннего цикла
    return 0;
}

```

#### Замечание

Внешний цикл определяет количество строк, выводимых на экран. Обратите внимание на то, что в блоке модификации данного цикла стоят два оператора. Первый, `++i`, будет увеличивать значение `i` на единицу после каждого выполнения внутреннего цикла, а второй, `cout << endl`, будет переводить выходной поток на новую строку. Внутренний цикл является телом внешнего цикла. Внутренний цикл определяет, сколько чисел нужно вывести в каждой строке, а в теле внутреннего цикла выводится нужное число.

Рассмотрим еще один пример использования вложенных циклов, который позволит вывести на экран следующую таблицу:

1				
1	3			
1	3	5		
1	3	5	7	
1	3	5	7	9

```

#include <iostream>
using namespace std;
int main()
{
    for (int i = 1; i <= 5; ++i, cout << endl) // внешний цикл
        for (int j = 1; j <= 2 * i - 1; j += 2) // внутренний цикл
            cout << j << "\t"; // тело внутреннего цикла
    return 0;
}

```

#### Замечание

---

В данном случае таблица состоит из пяти строчек, в каждой из которых печатаются только нечетные числа. Причем последнее нечетное число в строчке зависит от ее номера. Эта зависимость выражается через формулу  $k = 2i - 1$  (зависимость проверить самостоятельно), где  $k$  — последнее число в строке,  $i$  — номер текущей строки. Внешний цикл следит за номером текущей строки  $i$ , а внутренний цикл будет печатать нечетные числа из диапазона от 1 до  $2i - 1$ .

---

### 3.5. Примеры использования операторов цикла при решении задач

**1.** Написать программу, которая выводит на экран квадраты всех целых чисел от  $A$  до  $B$  ( $A$  и  $B$  — целые числа, при этом  $A \leq B$ ).

*Указания по решению задачи.* Необходимо перебрать все целые числа из интервала от  $A$  до  $B$ . Эти числа представляют собой упорядоченную последовательность, в которой каждое число отличается от предыдущего на единицу. При решении данной задачи можно использовать любой оператор цикла.

```

#include <iostream>                                #include <iostream>
using namespace std;                                using namespace std;
int main()                                         int main()
{
    int a, b;                                       {
    cout << "a = ";                                 int a, b;
    cin >> a;                                       cout << "a = ";
    cout << "b = ";                                 cin >> a;
    cin >> b;                                       cout << "b = ";
    int i = a;                                       cin >> b;
    while (i <= b)                                 int i = a;
        cout << i * i++ << "\t";                      do
    return 0;                                         cout << i * i++ << "\t";
}                                                 while (i <= b);
                                                return 0;
}

```

#### Замечание

---

Рассмотрим выражение  $i * i++$ , значение которого выводится на экран в теле каждого из циклов. С учетом приоритетов операций (см. приложе-

ние 3) вначале выполнится операция умножения, результат которой будет помещен в выходной поток, а затем постфиксный инкремент увеличит значение  $i$  на единицу.

---

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    for (int i = a; i <= b; i++)
        cout << i * i << "\t";
    return 0;
}
```

Три программы дадут одинаковый результат:

a	b	ответ
3	9	9 16 25 36 49 64 81

**2.** Написать программу, которая выводит на экран квадраты всех четных чисел из диапазона от  $A$  до  $B$  ( $A$  и  $B$  — целые числа, при этом  $A \leq B$ ).

*Указания по решению задачи.* Из диапазона целых чисел от  $A$  до  $B$  необходимо выбрать только четные числа. Напомним, что четными называются числа, которые делятся на два без остатка. Кроме того, четные числа представляют собой упорядоченную последовательность, в которой каждое число отличается от предыдущего на два. Решить эту задачу можно с помощью каждого оператора цикла.

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, i;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    i = (a%2)? a + 1: a;
    while (i <= b)
    {
        cout << i * i << "\t";
        i += 2
    }
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, i;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    i = (a%2)? a + 1 : a;
    do
        cout << i * i << "\t";
        i += 2;
    while (i <= b);
    return 0;
}
```

### Замечание

Начальное значение переменной  $i$  определяется с помощью тернарной операции. Если  $a$  — нечетное число, то при делении на 2 мы получим оста-

ток 1, который трактуется компилятором как *истина*, и, следовательно, переменной  $i$  будет присвоено значение  $a + 1$ . Если же  $a$  — четное число, то при делении на 2 мы получим остаток 0, который трактуется компилятором как *ложь*, и, следовательно, переменной  $i$  будет присвоено значение  $a$ . В результате, независимо от значения переменной  $a$ , переменной  $i$  будет присвоено четное значение. Тогда, увеличивая значение  $a$  на 2, мы всегда будем получать только четные числа.

---

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    for (int i = (a%2)? a+1: a; i <= b; i +=2)
        cout<< i * i<< "\t";
    return 0;
}
```

Три программы дадут нам одинаковый результат:

a	b	ответ
3	9	16 36 64

3. Построить таблицу значений функции  $y(x) = \frac{1}{x} + \sqrt{x+1}$  для  $x \in [a, b]$

с шагом  $h$ . Если в некоторой точке  $x$  функция не определена, то вывести на экран сообщение об этом.

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

/*Вспомогательная функция: выводит на экран значение функции в точке
x или сообщение о том, что функция не определена*/
void f(float x)
{
    if (!x || x < -1)
        cout << "функция не определена";
    else cout << 1/x+sqrt(x+1);
}

int main() // главная функция
{
    float a, b, h, x;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin>>b;
```

```

cout << "h = ";
cin >> h;
cout << "x\tf(x)\n"; // выводим заголовок таблицы
// перебираем все числа из отрезка [a, b] с шагом h
for (x = a; x <= b; x += h)
{
    cout << x << "\t"; // выводим на экран значение x
    f(x); // выводим на экран значение функции в точке x
    cout << endl; // переводим выходной поток на новую строку
}
return 0;
}

```

Результат работы программы для  $a = -2$ ,  $b = 4$ ,  $h = 2$ :

x	f(x)
-2	функция не определена
0	функция не определена
2	2.23205
4	2.48607

#### 4. Построить таблицу значений функции

$$y(x) = \begin{cases} (x^3 + 1)^2, & \text{при } x < 0; \\ 0, & \text{при } 0 \leq x < 1; \\ |x^2 - 5x + 1|, & \text{при } x \geq 1 \end{cases}$$

для  $x \in [a, b]$  с шагом  $h$ .

```

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

// вспомогательная функция: возвращает значение функции в точке x
float f(float x)
{
    if (x < 0) // условие первой ветви
        return pow(pow(x, 3) + 1, 2);
    else if (x < 1) // условие второй ветви
        return 0;
    else return fabs(x * x - 5 * x + 1); // по умолчанию третья ветвь
}

int main() // главная функция
{
    float a, b, h, x;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "h = ";
    cin >> h;
    cout << "x\tf(x)\n"; // выводим заголовок таблицы
}

```

```

// перебираем все числа из отрезка [a, b] с шагом h
for (x = a; x <= b; x += h)
    // выводим на экран значение x и значение функции в точке x
    cout << x << "\t" << f(x) << endl;
return 0;
}

```

Результат работы программы для  $a = -3$ ,  $b = 3$ ,  $h = 1.5$ :

x	f(x)
-3	676
-1.5	5.64063
0	0
1.5	4.25
3	5

### 3.6. Операторы безусловного перехода

В C++ есть четыре оператора, изменяющие естественный порядок выполнения операторов: оператор безусловного перехода **goto**, оператор выхода **break**, оператор перехода к следующей итерации цикла **continue**, оператор возврата из функции **return**.

**Оператор безусловного перехода goto.** Оператор безусловного перехода **goto** имеет формат

```
goto <метка>;
```

В теле той же функции должна присутствовать ровно одна конструкция вида

```
<метка>: <оператор>;
```

Оператор **goto** передает управление на помеченный меткой оператор. Рассмотрим пример использования оператора **goto**:

```

#include <iostream>
using namespace std;
int main()
{
    float x;
    metka: cout << "x = "; // метка
        cin >> x;
        if (x)
            cout << "y = " << 1/x << endl;
        else
        {
            cout << "функция не определена\n";
            goto metka; // передача управления метке
        }
    return 0;
}

```

## Замечание

В данном примере при попытке ввести 0 на экран будет выведено сообщение «функция не определена», после чего управление будет передано метке, и программа повторно попросит ввести значение *x*.

Следует учитывать, что использование оператора **goto** затрудняет чтение больших по объему программ, поэтому использовать его нужно только в крайних случаях.

**Оператор выхода break.** Этот оператор используется внутри операторов ветвления и цикла для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится **break**.

В параграфе 3.2 оператор **break** применялся для выхода из оператора **switch**, аналогичным образом он может применяться для выхода из других операторов.

**Оператор перехода к следующей итерации цикла continue.** Оператор перехода к следующей итерации цикла **continue** пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации (повторение тела цикла). Рассмотрим применение оператора **continue** на примере.

```
#include <iostream>
using namespace std;
int main()
{
    for (int i = 1; i < 100; ++i) // перебираем все числа от 1 до 99
    {
        if (i % 2) // если число нечетное,
                    // то переходим к следующей итерации
            continue;
        cout << i << "\t"; // выводим число на экран
    }
    return 0;
}
```

## Замечание

В результате работы данной программы на экран будут выведены только четные числа из интервала от 1 до 100, так как для нечетных чисел текущая итерация цикла прерывалась и команда

```
cout << i << "\t"
```

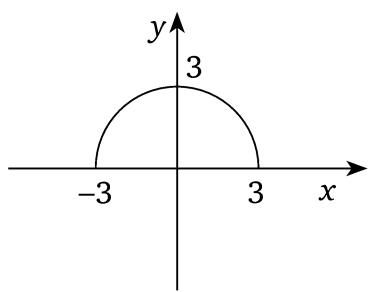
не выполнялась.

**Оператор возврата из функции return.** Оператор **return** завершает выполнение функции и передает управление в точку ее вызова. Данный оператор мы неоднократно использовали при разработке функций, возвращающих значение.

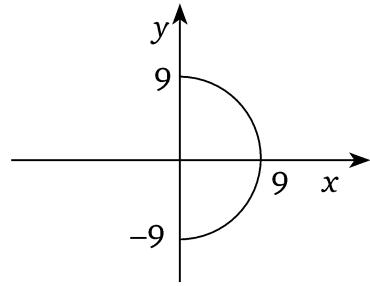
## Упражнения

I. Данна точка на плоскости с координатами  $(x, y)$ . Составить программу, которая выдает одно из сообщений «Да», «Нет», «На границе» в зависимости от того, лежит ли точка внутри заштрихованной области, вне заштрихованной области или на ее границе. Области задаются графически следующим образом:

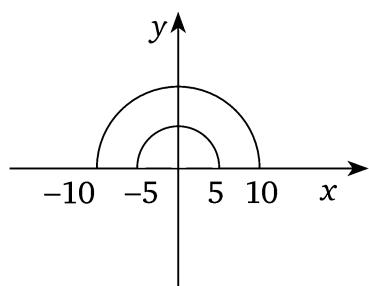
1.



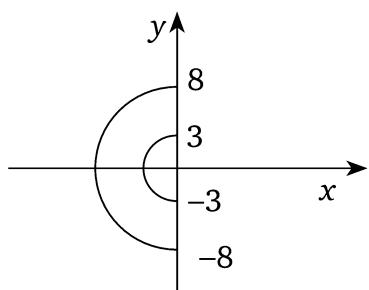
2.



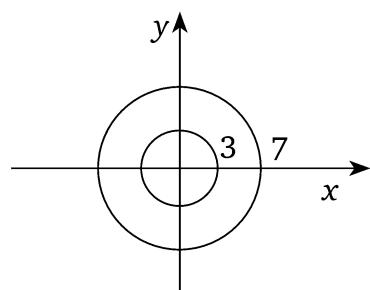
3.



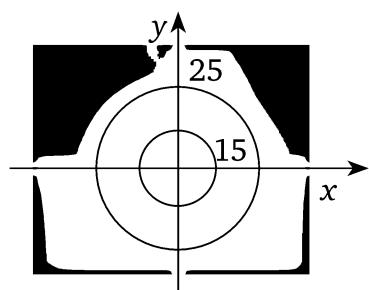
4.



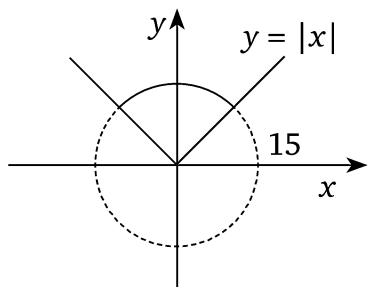
5.



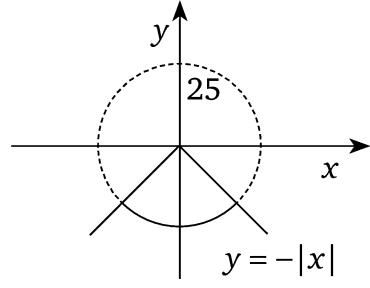
6.



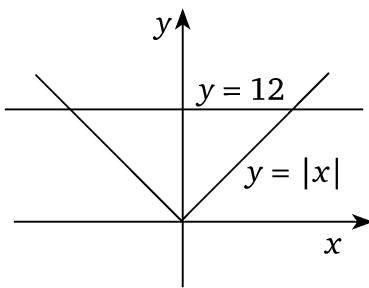
7.



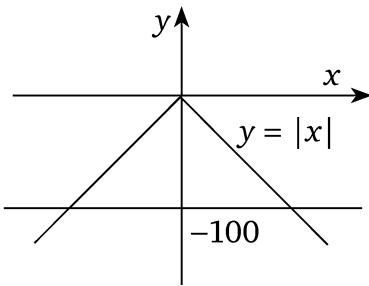
8.



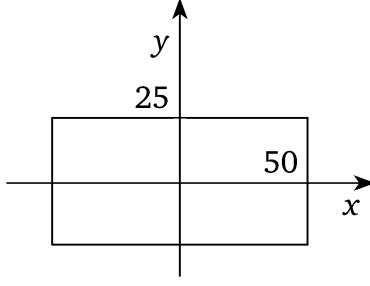
9.



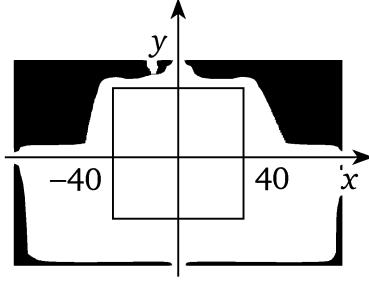
10.



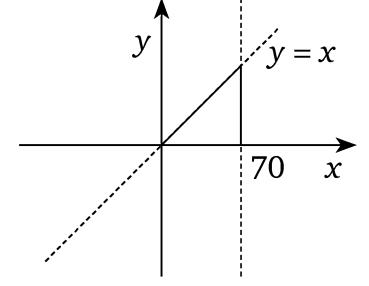
11.



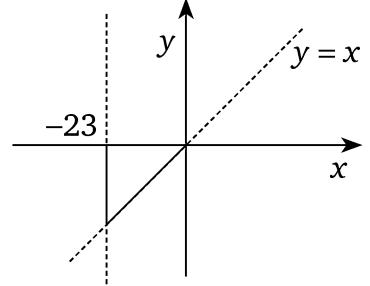
12.



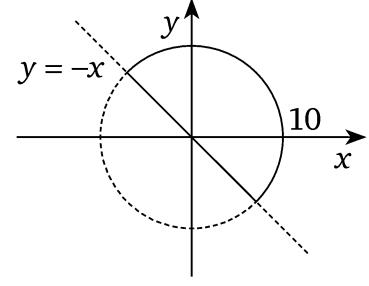
13.



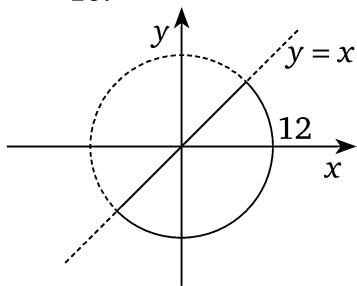
14.



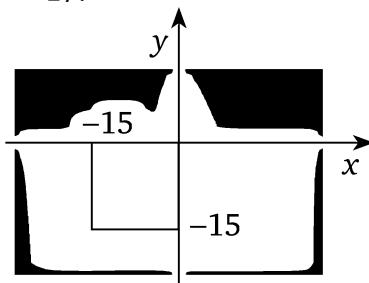
15.



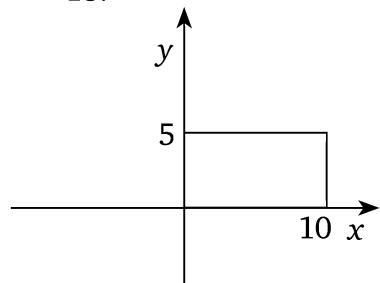
16.



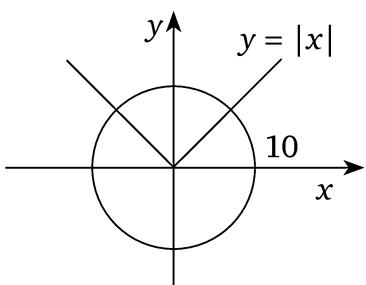
17.



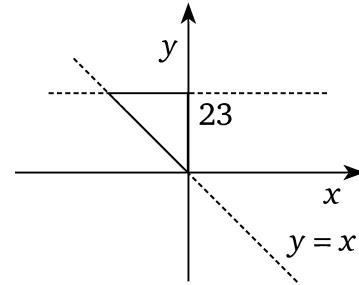
18.



19.



20.



## II. Составить программы.

1. Дан порядковый номер месяца, вывести на экран его название.
2. Дан порядковый номер дня месяца, вывести на экран количество дней, оставшихся до конца месяца.
3. Дан номер масти  $m$  ( $1 \leq m \leq 4$ ), определить название масти. Масти нумеруются: «пики» — 1, «трефы» — 2, «бубны» — 3, «червы» — 4.
4. Дан номер карты  $k$  ( $6 \leq k \leq 14$ ), определить достоинство карты. Достоинства определяются по следующему правилу: «туз» — 14, «король» — 13, «дама» — 12, «валет» — 11, «десятка» — 10, ..., «шестерка» — 6.
5. Дан номер масти  $m$  ( $1 \leq m \leq 4$ ) и номер достоинства карты  $k$  ( $6 \leq k \leq 14$ ). Определить полное название соответствующей карты в виде «дама пик», «шестерка бубен» и т.д.
6. С 1 января 1990 г. по некоторый день прошло  $t$  месяцев, определить название текущего месяца.
7. Дано расписание приемных часов врача. Вывести на экран приемные часы врача в заданный день недели (расписание придумать самостоятельно).
8. Проведен тест, оцениваемый в целочисленный баллах от нуля до 100. Вывести на экран оценку тестируемого в зависимости от набранного количества баллов: от 90 до 100 — «отлично», от 70 до 89 — «хорошо», от 50 до 69 — «удовлетворительно», менее 50 — «неудовлетворительно».
9. Даны два числа и арифметическая операция. Вывести на экран результат этой операции.
10. Дан год. Вывести на экран название животного, символизирующего этот год по восточному календарю.
11. Дан возраст человека мужского пола в годах. Вывести на экран возрастную категорию: до года — «младенец», от года до 11 лет — «ребенок», от 12 до 15 лет — «подросток», от 16 до 25 лет — «юноша», от 26 до 70 лет — «мужчина», более 70 лет — «пожилой человек».
12. Дан пол человека: м — мужчина, ж — женщина. Вывести на экран возможные мужские и женские имена в зависимости от введенного пола.
13. Дан признак транспортного средства: а — автомобиль, в — велосипед, м — мотоцикл, с — самолет, п — поезд. Вывести на экран максимальную скорость транспортного средства в зависимости от введенного признака.

14. Дан номер телевизионного канала (от 1 до 5). Вывести на экран наиболее популярные программы заданного канала.

15. Дан признак геометрической фигуры на плоскости: к — круг, п — прямоугольник, т — треугольник. Вывести на экран периметр и площадь заданной фигуры (данные, необходимые для расчетов, запросить у пользователя).

### III. Вывести на экран:

#### Замечание

Решите каждую задачу тремя способами: используя операторы цикла *while*, *do while* и *for*.

- 1) целые числа 1, 3, 5, ..., 21 в строчку через пробел;
- 2) целые числа 10, 12, 14, ..., 60 в обратном порядке в столбик;
- 3) числа следующим образом:

10 10.4

11 11.4

...

25 25.4

- 4) числа следующим образом:

25 25.5 24.8

26 26.5 25.8

...

35 35.5 34.8

5) таблицу соответствия между весом в фунтах и весом в килограммах для значений 1, 2, 3, ..., 10 фунтов (1 фунт = 453 г);

6) таблицу перевода 5, 10, 15, ..., 120 долларов США в рубли по текущему курсу (значение курса вводится с клавиатуры);

7) таблицу стоимости для 10, 20, 30, ..., 100 шт. товара, при условии, что одна штука товара стоит  $x$  руб. (значение  $x$  вводится с клавиатуры);

8) таблицу перевода расстояний в дюймах в сантиметры для значений 2, 4, 6, ..., 12 дюймов (1 дюйм = 25,4 мм);

9) кубы всех целых чисел из диапазона от  $A$  до  $B$  ( $A \leq B$ ) в обратном порядке;

10) все целые числа из диапазона от  $A$  до  $B$  ( $A \leq B$ ), оканчивающиеся на цифру  $X$ ;

11) все целые числа из диапазона от  $A$  до  $B$  ( $A \leq B$ ), оканчивающиеся на цифру  $X$  или  $Y$ ;

12) все целые числа из диапазона от  $A$  до  $B$  ( $A \leq B$ ), оканчивающиеся на любую четную цифру;

13) только положительные целые числа из диапазона от  $A$  до  $B$  ( $A \leq B$ );

14) все целые числа из диапазона от  $A$  до  $B$ , кратные трем ( $A \leq B$ );

15) все четные числа из диапазона от  $A$  до  $B$ , кратные трем ( $A \leq B$ );

16) только отрицательные четные числа из диапазона от  $A$  до  $B$  ( $A \leq B$ );

17) все двухзначные числа, в записи которых все цифры разные;

18) все двухзначные числа, в которых старшая цифра отличается от младшей не больше чем на единицу;

19) все трехзначные числа, которые начинаются и заканчиваются на одну и ту же цифру;

20) все трехзначные числа, в которых хотя бы две цифры повторяются.

**IV. Вывести на экран числа в виде следующих таблиц:**

1.	5	5	5	5	5	5	2.	1	2	3	...	10
	5	5	5	5	5	5		1	2	3	...	10
	5	5	5	5	5	5		1	2	3	...	10
	5	5	5	5	5	5		1	2	3	...	10
3.	-10	-9	-8	...	12		4.	41	42	43	...	50
	-10	-9	-8	...	12			51	52	53	...	60
	-10	-9	-8	...	12			61	62	63	...	70
	-10	-9	-8	...	12			...				
	-10	-9	-8	...	12			71	72	73	...	80
5.	5						6.	1	1	1	1	1
	5	5						1	1	1	1	
	5	5	5					1	1	1		
	5	5	5	5				1	1			
	5	5	5	5	5			1				
7.	1						8.	6	6	6	6	6
	2	2						7	7	7	7	
	3	3	3					8	8	8		
	4	4	4	4				9	9			
	5	5	5	5	5			10				
9.	7						10.	8	8	8	8	8
	6	6						7	7	7	7	
	5	5	5					6	6	6		
	4	4	4	4				5	5			
	3	3	3	3	3			4				
11.	1						12.	1				
	1	2						2	1			
	1	2	3					3	2	1		
	1	2	3	4				4	3	2	1	
	1	2	3	4	5			5	4	3	2	1
13.	0	1	2	3	4		14.	4	3	2	1	0
	0	1	2	3				3	2	1	0	
	0	1	2					2	1	0		
	0	1						1	0			
	0							0				
15.	1						16.	8				
	0							7				
	2	2						7	7			
	0	0						6	6			
	3	3	3					6	6	6		
	0	0	0					5	5	5		
	4	4	4	4				5	5	5	5	
	0	0	0	0				4	4	4	4	
	5	5	5	5	5							
	0	0	0	0	0							

17.	1		18.	9	
	6			4	
2	2		8	8	
7	7		3	3	
3	3	3	7	7	7
8	8	8	2	2	2
4	4	4	4	6	6
9	9	9	9	1	1
19.	3		20.	2	2
0				3	4
2	3			2	2
9	0			2	3
2	2	3		2	2
8	9	0		1	2
2	2	2	3	2	2
7	8	9	0	0	1
2	2	2	2	2	
6	7	8	9	0	-1

V. Построить таблицу значений функции  $y = f(x)$  для  $x \in [a, b]$  с шагом  $h$ . Если в некоторой точке  $x$  функция не определена, вывести на экран сообщение об этом.

#### Замечание

Для решения задачи использовать вспомогательную функцию.

$$\begin{array}{lll}
 1. \quad y = \frac{1}{(1+x)^2}; & 2. \quad y = \frac{1}{x^2-1}; & 3. \quad y = \sqrt{x^2-1}; \\
 4. \quad y = \sqrt{5-x^3}; & 5. \quad y = \ln(x-1); & 6. \quad y = \ln(4-x^2); \\
 7. \quad y = \frac{x}{\sqrt{2x-1}}; & 8. \quad y = \frac{3x+4}{\sqrt{x^2+2x+1}}; & 9. \quad y = \frac{1}{x-1} + \frac{2}{1-4x}; \\
 10. \quad y = \ln|x-2|; & 11. \quad y = \ln \frac{x}{x-2}; & 12. \quad y = \ln(x^4-1) \ln(1+x); \\
 13. \quad y = \frac{\ln(x-2)}{\sqrt{5x+1}}; & 14. \quad y = \frac{\sqrt{x^2-2x+1}}{\ln(4-2x)}; & 15. \quad y = \ln|3x| \sqrt{2x^5-1}; \\
 16. \quad y = \frac{3}{|x^3+8|}; & 17. \quad y = \frac{x+4}{x^2-2} + \sqrt{x^3-1}; & 18. \quad y = \sqrt{x^2+1} - \sqrt{x^2+5}; \\
 19. \quad y = \frac{\sqrt{x^3-1}}{\sqrt{x^2-1}}; & 20. \quad y = \frac{1}{x+7} + \ln(1-|x|). &
 \end{array}$$

VI. Построить таблицу значений функции  $y = f(x)$  для  $x \in [a, b]$  с шагом  $h$ .

#### Замечание

Для решения задачи использовать вспомогательную функцию.

$$1. \quad y = \begin{cases} \frac{1}{(0.1+x)^2}, & \text{если } x \geq 0.9; \\ 0.2x + 0.1, & \text{если } 0 \leq x < 0.9; \\ x^2 + 0.2, & \text{если } x < 0. \end{cases}$$

$$3. \quad y = \begin{cases} 0, & \text{если } x < a; \\ \frac{x-a}{x+a}, & \text{если } x > a; \\ 1, & \text{если } x = a. \end{cases}$$

$$5. \quad y = \begin{cases} a+b, & \text{если } x^2 - 5x < 0; \\ a-b, & \text{если } 0 \leq (x^2 - 5x) < 10; \\ ab, & \text{если } x^2 - 5x \geq 10. \end{cases}$$

$$7. \quad y = \begin{cases} -4, & \text{если } x < 0; \\ x^2 + 3x + 4, & \text{если } 0 \leq x < 1; \\ 2, & \text{если } x \geq 1. \end{cases}$$

$$9. \quad y = \begin{cases} (x^2 - 1)^2, & \text{если } x < 1; \\ \frac{1}{(1+x)^2}, & \text{если } x > 1; \\ 0, & \text{если } x = 1. \end{cases}$$

$$11. \quad y = \begin{cases} x^2 + 5, & \text{если } x \leq 5; \\ 0, & \text{если } 5 < x < 20; \\ 1, & \text{если } x \geq 20. \end{cases}$$

$$13. \quad y = \begin{cases} 1, & \text{если } x = 1 \text{ или } x = -1; \\ \frac{-1}{1-x}, & \text{если } x \geq 0 \text{ и } x \neq 1; \\ \frac{1}{1+x}, & \text{если } x < 0 \text{ и } x \neq -1. \end{cases}$$

$$15. \quad y = \begin{cases} 1, & \text{если } (x-1) < 1; \\ 0, & \text{если } (x-1) = 1; \\ -1, & \text{если } (x-1) > 1. \end{cases}$$

$$17. \quad y = \begin{cases} a+bx, & \text{если } x < 93; \\ b-ac, & \text{если } 93 \leq x \leq 120; \\ abx, & \text{если } x > 120. \end{cases}$$

$$19. \quad b_1 = 1, \quad b_2 = 5, \quad b_{2n} = b_{2n-1} + b_{2n-2}, \\ b_{2n+1} = b_{2n} - b_{2n-1}.$$

$$2. \quad y = \begin{cases} \sin(x), & \text{если } |x| < 3; \\ \frac{\sqrt{x^2 + 1}}{\sqrt{x^2 + 5}}, & \text{если } 3 \leq |x| < 9; \\ \sqrt{x^2 + 1} - \sqrt{x^2 + 5}, & \text{если } |x| \geq 9. \end{cases}$$

$$4. \quad y = \begin{cases} x^3 - 0.1, & \text{если } |x| \leq 0.1; \\ 0.2x - 0.1, & \text{если } 0.1 < |x| \leq 0.2; \\ x^3 + 0.1, & \text{если } |x| > 0.2. \end{cases}$$

$$6. \quad y = \begin{cases} x^2, & \text{если } (x^2 + 2x + 1) < 2; \\ \frac{1}{x^2 - 1}, & \text{если } 2 \leq (x^2 + 2x + 1) < 3; \\ 0, & \text{если } (x^2 + 2x + 1) \geq 3. \end{cases}$$

$$8. \quad y = \begin{cases} x^2 - 1, & \text{если } |x| \leq 1; \\ 2x - 1, & \text{если } 1 < |x| \leq 2; \\ x^5 - 1, & \text{если } |x| > 2. \end{cases}$$

$$10. \quad y = \begin{cases} x^2, & \text{если } (x+2) \leq 1; \\ \frac{1}{x+2}, & \text{если } 1 < (x+2) < 10; \\ x+2, & \text{если } (x+2) \geq 10. \end{cases}$$

$$12. \quad y = \begin{cases} 0, & \text{если } x < 0; \\ x^2 + 1, & \text{если } x \geq 0 \text{ и } x \neq 1; \\ 1, & \text{если } x = 1. \end{cases}$$

$$14. \quad y = \begin{cases} 0.2x^2 - x - 0.1, & \text{если } x < 0; \\ \frac{x^2}{x-0.1}, & \text{если } x > 0 \text{ и } x \neq 0.1; \\ 0, & \text{если } x = 0.1. \end{cases}$$

$$16. \quad y = \begin{cases} x, & \text{если } x > 0; \\ 0, & \text{если } -1 \leq x \leq 0; \\ x^2, & \text{если } x < -1. \end{cases}$$

$$18. \quad y = \begin{cases} x^2 - 0.3, & \text{если } y < 3; \\ 0, & \text{если } 3 \leq x \leq 5; \\ x^2 + 1, & \text{если } x > 5. \end{cases}$$

$$20. \quad y = \begin{cases} \sin x, & \text{если } |x| < \frac{\pi}{2}; \\ \cos x, & \text{если } \frac{\pi}{2} \leq |x| \leq \pi; \\ 0, & \text{если } |x| > \pi. \end{cases}$$

# Глава 4

## РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ

### 4.1. Вычисление членов рекуррентной последовательности

Пусть  $a_1, a_2, \dots, a_n$  — произвольная числовая последовательность. **Рекуррентным соотношением** называется такое соотношение между членами последовательности, в котором каждый следующий член выражается через несколько предыдущих, т.е

$$a_k = f(a_{k-1}, a_{k-2}, \dots, a_{k-l}), k > l. \quad (4.1)$$

Последовательность задана рекуррентно, если для нее определено рекуррентное соотношение вида (4.1) и заданы первые  $l$  ее членов.

Самым простым примером рекуррентной последовательности является арифметическая прогрессия. Рекуррентное соотношение для нее записывается в виде  $a_k = a_{k-1} + d$ , где  $d$  — разность прогрессии. Зная первый элемент и разность прогрессии и используя данное рекуррентное соотношение, можно последовательно вычислить все остальные члены прогрессии.

Рассмотрим пример программы, в которой вычисляются первые  $n$  членов арифметической прогрессии при условии, что  $a_1 = \frac{1}{2}$  и  $d = \frac{1}{4}$ .

```
#include <iostream>
using namespace std;
int main()
{
    // задаем первый член последовательности и разность прогрессии
    float a = 0.5, d = 0.25;
    int n;
    cout << "n = ";
    cin >> n; // ввели количество членов последовательности
    cout << "a1: " << a << endl; // вывели первый член
    // организуем вычисление 2 ... n члена последовательности
    for (int i = 2; i <= n; i++)
    {
        // для этого прибавляем к предыдущему члену значение d
        a = a + d;
        // и выводим новое значение a на экран
        cout << "a" << i << " = " << a << endl;
    }
    return 0;
}
```

Результат работы программы:

```
n    состояния экрана
5    a1: 0.5
     a2: 0.75
     a3: 1.
     a4: 1.25
     a5: 1.5
```

Другим примером рекуррентной последовательности является геометрическая прогрессия. Рекуррентное соотношение для нее записывается в виде  $b_k = b_{k-1} \cdot q$ , где  $q$  — знаменатель прогрессии. Рассмотрим пример программы, в которой вычисляются первые  $n$  членов арифметической прогрессии при условии, что  $b_1 = 1$ ,  $q = 2$ .

```
#include <iostream>
using namespace std;
int main()
{
    // задали первый член последовательности и знаменатель прогрессии
    int b = 1, q = 2, n;
    cout << "n = ";
    cin >> n; // ввели количество членов последовательности
    cout << "b1 = " << b << endl; // вывели первый член
                                    // последовательности
    // организуем вычисление 2 ... n члена последовательности
    for (int i = 2; i <= n; i++)
    {
        // для этого умножаем предыдущее значение b на q
        b = b * q;
        // и выводим новое значение b на экран
        cout << "b" << i << " = " << b << endl;
    }
    return 0;
}
```

Результат работы программы:

```
n    состояния экрана
5    b1: 1
     b2: 2
     b3: 4
     b4: 8
     b5: 16
```

В арифметической и геометрической прогрессиях каждый член последовательности зависит только от одного предыдущего значения. Более сложная зависимость представлена в последовательности Фибоначчи:  $a_1 = a_2 = 1$ ,  $a_n = a_{n-1} + a_{n-2}$ . В этом случае каждый член последовательности зависит от значений двух предыдущих членов. Рассмотрим пример программы, в которой вычисляются первые  $n$  членов последовательности Фибоначчи.

```

#include <iostream>
using namespace std;
int main()
{
    // задали первый и второй члены последовательности Фибоначчи
    int a1 = 1, a2 = 1, a, n;
    cout << "n = ";
    cin >> n; // ввели количество членов последовательности
    // выводим известные члены
    cout << "a1 = " << a1 << endl << "a2 = " << a2 << endl;
    /*Организуем цикл для вычисления членов последовательности с номерами 3, 4, ..., n. При этом в переменной a1 будет храниться значение члена последовательности с номером i-2, в переменной a2 - значение члена с номером i-1, переменная a будет использоваться для вычисления члена с номером i.*/
    for (int i = 3; i<= n; i++)
    {
        // по рекуррентному соотношению вычисляем член
        // последовательности с номером i и выводим
        // его значение на экран
        a = a1 + a2;
        cout << "a" << i << " = " << a << endl;
        // выполняем рекуррентный пересчет для следующего шага цикла
        // в элемент с номером i-2 записываем значение
        // элемента с номером i-1
        a1 = a2;
        // в элемент с номером i-1 записываем
        // значение элемента с номером i
        a2 = a;
    }
    return 0;
}

```

Результат работы программы:

```

n      состояние экрана
5      a1: 1
      a2: 1
      a3: 2
      a4: 3
      a5: 5

```

В рассмотренных случаях мы выводили на экран значения первых  $n$  элементов рекуррентной последовательности. Иногда нам бывает необходимо найти только значение  $n$ -го элемента последовательности. Для этого в программе нужно исключить вывод значения на каждом шаге цикла. В качестве примера рассмотрим программу, в которой вычисляется  $n$ -й элемент последовательности, заданной следующим образом:

$$b_1 = 1, \quad b_2 = 2, \quad b_n = \frac{b_{n-1} - b_{n-2}}{(n-1)^2}.$$

```

#include <iostream>
#include <cmath>

```

```

using namespace std;
int main()
{
    // задаем первый и второй элементы последовательности
    float b1 = 1, b2 = 2, b;
    int n;
    // вводим количество элементов последовательности
    cout << "n = "; cin >> n;
    /* Организуем цикл для вычисления элементов с номерами 3, 4, ..., n.
    При этом в переменной b1 будет храниться значение элемента последо-
    вательности с номером i-2, в переменной b2 - значение элемента с но-
    мером i-1, переменная b будет использоваться для вычисления элемента
    с номером i. */
    for (int i = 3; i <= n; i++)
    {
        // по рекуррентному соотношению вычисляем i-й элемент
        // последовательности и выполняем рекуррентный пересчет
        // для следующего шага цикла
        b = (b1 - b2)/pow(i-1.0,2);

        b1 = b2;
        b2 = b;
    }
    cout << "b" << n << " = " << b << endl; // выводим значение b
                                                // на экран
    return 0;
}

```

Результат работы программы:

```

n      состояние экрана
5      5 элемент: -0.03125

```

## Упражнения

Написать программу, вычисляющую первые  $n$  элементов заданной последовательности:

- 1)  $b_1 = 9, b_n = 0.1, b_{n-1} + 10;$
- 2)  $b_1 = -1, b_n = 9 - 2b_{n-1};$
- 3)  $b_1 = 1, b_n = 0.2b_{n-1}^4 + 1;$
- 4)  $b_1 = 4.7, b_n = \sin b_{n-1} + \pi;$
- 5)  $b_1 = 0.1, b_n = \frac{1}{6}(0.05 + b_{n-1}^3);$
- 6)  $b_1 = 2, b_n = 0.5\left(\frac{1}{b_{n-1}} + b_{n-1}\right);$
- 7)  $b_1 = 5, b_n = (-1)^n b_{n-1} - 8;$
- 8)  $b_1 = -1, b_2 = 1, b_n = 3b_{n-1} - 2b_{n-2};$
- 9)  $b_1 = -10, b_2 = 2, b_n = |b_{n-2}| - 6b_{n-1};$
- 10)  $b_1 = 2, b_2 = 4, b_n = 6b_{n-1} - b_{n-2};$
- 11)  $b_1 = 5, b_n = \frac{b_{n-1}}{n^2 + n + 1};$

$$12) b_1 = 0.5, b_2 = 0.2, b_{n+1} = b_n^2 + \frac{b_{n-1}}{n};$$

$$13) b_1 = 1, b_n = \frac{1}{4} \left( 3b_{n-1} + \frac{1}{3b_{n-1}} \right);$$

$$14) b_1 = 2, b_2 = 1, b_n = \frac{2}{3}b_{n-2} - \frac{1}{3}b_{n-1}^2;$$

$$15) b_1 = 1, b_2 = 2, b_n = \frac{b_{n-2}}{4} + \frac{5}{b_{n-1}^2};$$

$$16) b_1 = 1, b_2 = 2, b_n = \frac{nb_{n-2} - b_{n-1}}{n+1};$$

$$17) b_1 = 4, b_2 = 2, b_n = \frac{b_{n-2}}{n} + \frac{n^2}{b_{n-1}};$$

$$18) b_1 = 100, b_{2n} = b_{2n-1}/10, b_{2n+1} = b_{2n} + 10;$$

$$19) b_1 = 0, b_{2n} = b_{2n-1} + 3, b_{2n+1} = 2b_{2n};$$

$$20) b_1 = 1, b_2 = 5, b_{2n} = b_{2n-1} + b_{2n-2}, b_{2n+1} = b_{2n} - b_{2n-1}.$$

# Глава 5

## ВЫЧИСЛЕНИЕ КОНЕЧНЫХ И БЕСКОНЕЧНЫХ СУММ И ПРОИЗВЕДЕНИЙ

### 5.1. Вычисление конечных сумм и произведений

Решение многих задач связано с нахождением суммы или произведения элементов заданной последовательности. Здесь мы рассмотрим основные приемы вычисления конечных сумм и произведений.

Пусть  $u_1(x), u_2(x), \dots, u_n(x)$  — произвольная последовательность  $n$  функций. Будем рассматривать конечную сумму вида  $u_1(x) + u_2(x) + \dots + u_n(x)$ . Такую сумму можно записать более компактно, используя следующее обозначение:  $u_1(x) + u_2(x) + \dots + u_n(x) = \sum_{i=1}^n u_i(x)$ . При  $n \leq 0$  значение суммы равно 0.

В дальнейшем будем также использовать сокращенную запись для конечного произведения данной последовательности, которая выглядит следующим образом:  $u_1(x) \cdot u_2(x) \cdot \dots \cdot u_n(x) = \prod_{i=1}^n u_i(x)$ .

#### Пример 1

Написать программу, которая подсчитывает сумму натуральных чисел от 1 до  $n$  ( $n \geq 1$ ).

*Указания по решению задачи.* Пусть  $s_n$  — сумма натуральных чисел от 1 до  $n$ . Тогда  $s_n = 1 + 2 + \dots + (n - 1) + n = (1 + 2 + \dots + (n - 1)) + n = s_{n-1} + n$ ,  $s_0 = 0$ . Мы пришли к рекуррентному соотношению  $s_0 = 0$ ,  $s_n = s_{n-1} + n$ , которым мы можем воспользоваться для подсчета суммы. Соотношение  $s_n = s_{n-1} + n$  говорит о том, что сумма на  $n$ -ом шаге равна сумме, полученной на предыдущем шаге, плюс очередное слагаемое.

```
#include <iostream>
using namespace std;
int main()
{
    int n, s = 0;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++) // выполняем n шагов
        // и на каждом i-ом шаге
    s += i; // используем полученное рекуррентное соотношение
```

```

cout << "s = " << s << endl;
return 0;
}

```

Рассмотрим пошаговое выполнение программы.

Номер шага	Значение счетчика	Результат (значение $s$ )
1	$i = 1$	$0 + 1 = 1$
2	$i = 2$	$1 + 2 = 3$
3	$i = 3$	$1 + 2 + 3 = 6$
...	...	...
$n$	$i = n$	$1 + 2 + 3 + 4 + \dots + n$

### Пример 2

Написать программу, которая считает  $x^n$  для вещественного  $x$  и натурального  $n$ .

*Указание по решению задачи.* Из свойства степенной функции ( $x^0 = 1$ ,  $x^n = x^{n-1} \cdot x$ ) следует, что ее можно вычислять, используя рекуррентное соотношение  $b_0 = 1$ ,  $b_n = b_{n-1} \cdot x$ .

```

#include <iostream>
using namespace std;
int main()
{
    int n;
    float x, b = 1;
    cout << "x = ";
    cin >> x;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++)
        b *= x;
    cout << "x^n = " << b << endl;
    return 0;
}

```

Рассмотрим пошаговое выполнение программы:

Номер шага	Значение счетчика	Результат (значение $b$ )
1	$i = 1$	$1 \cdot x$
2	$i = 2$	$x \cdot x = x^2$
3	$i = 3$	$x^2 \cdot x = x^3$
...	...	...
$n$	$i = n$	$x^{n-1} \cdot x = x^n$

### Пример 3

Написать программу для подсчета суммы:  $-\sin x + \sin 2x - \sin 3x + \dots + (-1)^n \sin nx$  ( $n \geq 1$ ).

*Указания по решению задачи.* Если пронумеровать слагаемые, начиная с номера 1, то мы увидим закономерность: знак «минус» ставится перед слагаемыми с нечетными номерами, а знак «плюс» — с четными, при этом сумму можно выразить рекуррентным соотношением:  $s_0 = 0$ ,  $s_n = s_{n-1} + (-1)^n \sin(nx)$ . Введем вспомогательную переменную *singl*, которая будет использоваться для хранения знака очередного слагаемого. Так как первое слагаемое отрицательно, то начальное значение переменной *singl* равно  $-1$ . После каждой итерации цикла значение переменной *singl* будем заменять на противоположное с помощью унарного минуса. В этом случае рекуррентное соотношение можно преобразовать к виду  $s_0 = 0$ ,  $singl_0 = -1$ ,  $s_n = s_{n-1} + singl \cdot \sin(nx)$ ,  $singl_n = -singl_{n-1}$ .

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int n, singl = -1;
    float x, s = 0;
    cout << "x = ";
    cin >> x;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        s += singl * sin(i * x); // 1
        singl = -singl;          // 2
    }
    cout << "s = " << s << endl;
    return 0;
}
```

Рассмотрим пошаговое выполнение программы.

Номер шага	Значение счетчика	Результат (значение <i>s</i> )
1	$i = 1$	$0 - \sin x = \sin x$
2	$i = 2$	$-\sin x + \sin 2x$
3	$i = 3$	$-\sin x + \sin 2x - \sin 3x$
...	...	...
$n$	$i = n$	$-\sin x + \sin 2x - \sin 3x + \dots + (-1) \sin nx$

Подумайте, что произойдет, если поменять местами операторы 1 и 2.

#### Пример 4

Написать программу для подсчета суммы

$$S_n = \frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2} + \frac{\cos x + \cos 2x + \cos 3x}{3} + \dots + \frac{\cos x + \dots + \cos nx}{n},$$

где  $x$  — вещественное число,  $n$  — натуральное число.

*Указания по решению задачи.* Если пронумеровать слагаемые, начиная с 1, то мы увидим, что номер слагаемого совпадает со значением знаменателя.

теля. Рассмотрим каждый числитель отдельно:  $b_1 = \cos x$ ,  $b_2 = \cos x + \cos 2x$ ,  $b_3 = \cos x + \cos 2x + \cos 3x$ ... Эту последовательность можно представить рекуррентным соотношением

$$b_0 = 0, b_n = b_{n-1} + \cos nx. \quad (5.1)$$

Теперь сумму можно представить следующим образом:

$$S_n = \frac{b_1}{1} + \frac{b_2}{2} + \frac{b_3}{3} + \dots + \frac{b_n}{n},$$

а для нее справедливо рекуррентное соотношение

$$S_0 = 0, S_n = S_{n-1} + \frac{b_n}{n}. \quad (5.2)$$

При составлении программы будем использовать формулы (5.1) и (5.2).

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int n;
    float x, s = 0, b = 0;
    cout << "x = ";
    cin >> x;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        b += cos(i*x);
        s += b/i;
    }
    cout << "s = " << s << endl;
    return 0;
}
```

Рассмотрим пошаговое выполнение программы.

Номер шага	Значение счетчика	Значение $b$	Значение $s$
1	$i = 1$	$\cos x$	$\frac{\cos x}{1}$
2	$i = 2$	$\cos x + \cos 2x$	$\frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2}$
3	$i = 3$	$\cos x + \cos 2x + \cos 3x$	$\frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2} + \frac{\cos x + \cos 2x + \cos 3x}{3}$
...	...	...	...
$n$	$i = n$	$\cos x + \cos 2x + \cos 3x$	$\frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2} + \dots + \frac{\cos x + \dots + \cos nx}{n}$

## Пример 5

Написать программу для подсчета суммы

$$S_n = \sum_{i=1}^n \frac{(-1)^{i+1} x^i}{i!},$$

где  $x$  — вещественное число;  $n$  — натуральное число.

*Указания по решению задачи.* Переайдем от сокращенной формы записи к развернутой, получим

$$S_n = \frac{x}{1!} - \frac{x^2}{2!} + \frac{x^3}{3!} - \dots + \frac{(-1)^{n+1} x^n}{n!}.$$

Каждое слагаемое формируется по формуле

$$a_n = \frac{(-1)^{n+1} x^n}{n!}.$$

Если в эту формулу подставить  $n = 0$ , то получим

$$a_0 = \frac{(-1)^1 x^0}{0!} = -1.$$

Запись  $n!$  читается как « $n$  факториал». По определению факториала:  $0! = 1! = 1$ ,  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ ,  $n! = (n-1)!n$ . Таким образом, факториал выражается рекуррентным соотношением.

Чтобы не вводить несколько рекуррентных соотношений (отдельно для числителя и для знаменателя), выразим последовательность слагаемых рекуррентным соотношением вида  $a_n = a_{n-1}q$ , где  $q$  нам пока не известно. Найти его можно из выражения  $q = \frac{a_n}{a_{n-1}}$ . Произведя расчеты, мы получим,

что  $q = -\frac{x}{i}$ . Следовательно, для последовательности слагаемых мы получили рекуррентное соотношение

$$a_0 = -1, a_i = -a_{i-1} \cdot \frac{x}{i}, \quad (5.3)$$

а всю сумму, по аналогии с предыдущими примерами, можно представить рекуррентным соотношением

$$S_0 = 0, S_n = S_{n-1} + a_n. \quad (5.4)$$

Таким образом, при составлении программы будем пользоваться формулами (5.3) и (5.4).

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    float x, s = 0, a = -1;
    cout << "x = ";
    cin >> x;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++)
```

```

{
    a *= -x/i;
    s += a;
}
cout << "s = " << s << endl;
return 0;
}

```

Рассмотрим пошаговое выполнение программы:

Номер шага	Значение счетчика	Значение $a$	Значение $s$
1	$i = 1$	$-(-1)\frac{x}{1} = \frac{x^1}{1!}$	$0 + \frac{x^1}{1!} = \frac{x^1}{1!}$
2	$i = 2$	$-\frac{x}{1} \cdot \frac{x}{2} = -\frac{x^2}{2!}$	$\frac{x^1}{1!} - \frac{x^2}{2!}$
3	$i = 3$	$-(-\frac{x^2}{2!}) \cdot \frac{x}{3} = \frac{x^3}{3!}$	$\frac{x^1}{1!} - \frac{x^2}{2!} + \frac{x^3}{3!}$
...	...	...	...
$n$	$i = n$	$-\frac{(-1)^n x^{n-1}}{(n-1)!} \cdot \frac{x^n}{n} = \frac{(-1)^{n+1} x^n}{n!}$	$\frac{x^1}{1!} - \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{(-1)^{n+1} x^n}{n!}$

### Пример 6

Написать программу для подсчета произведения

$$P_k = \prod_{n=1}^k \left( 1 + \frac{x^{2n} + x^n}{n} \right),$$

где  $x$  — вещественное число,  $n$  — натуральное число.

*Указания по решению задачи.* Преобразуем заданное выражение к виду

$$P_k = \prod_{n=1}^k \left[ 1 + \frac{x^n(x^n + 1)}{n} \right]$$

и перейдем от сокращенной формы записи к развернутой:

$$P_k = \left[ 1 + \frac{x^1(x^1 + 1)}{1} \right] \left[ 1 + \frac{x^2(x^2 + 1)}{2} \right] \dots \left[ 1 + \frac{x^k(x^k + 1)}{k} \right].$$

В числителе каждой дроби встречается  $x^n$  (см. пример 2), его можно вычислить по рекуррентному соотношению

$$b_0 = 1, b_n = b_{n-1} \cdot x. \quad (5.5)$$

Тогда произведение можно представить как

$$P_k = \left[ 1 + \frac{b_1(b_1 + 1)}{1} \right] \left[ 1 + \frac{b_2(b_2 + 1)}{2} \right] \dots \left[ 1 + \frac{b_k(b_k + 1)}{k} \right],$$

что в свою очередь можно выразить рекуррентным соотношением

$$P_0 = 1, P_k = P_{k-1} \left[ 1 + \frac{b_k(b_k + 1)}{k} \right]. \quad (5.6)$$

При составлении программы будем пользоваться формулами (5.5) и (5.6).

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    float x, p = 1, b = 1;
    cout << "x = ";
    cin >> x;
    cout << "n = ";
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        b *= x;
        p *= (1 + b*(b + 1)/i);
    }
    cout << "p = " << p << endl;
    return 0;
}
```

Рассмотрим пошаговое выполнение программы:

Номер шага	Значение счетчика	Значение $b$	Значение $p$
1	$i = 1$	$x$	$1 \cdot \left[ 1 + \frac{x(x+1)}{1} \right]$
2	$i = 2$	$x^2$	$1 \cdot \left[ 1 + \frac{x(x+1)}{1} \right] \left[ 1 + \frac{x^2(x^2+1)}{2} \right]$
3	$i = 3$	$x^3$	$1 \cdot \left[ 1 + \frac{x(x+1)}{1} \right] \left[ 1 + \frac{x^2(x^2+1)}{2} \right] \left[ 1 + \frac{x^3(x^3+1)}{3} \right]$
...	...	...	...
$n$	$i = k$	$x^k$	$1 \cdot \left[ 1 + \frac{x(x+1)}{1} \right] \left[ 1 + \frac{x^2(x^2+1)}{2} \right] \dots \left[ 1 + \frac{x^k(x^k+1)}{k} \right]$

---

## 5.2. Вычисление бесконечных сумм

Будем теперь рассматривать бесконечную сумму вида

$$u_1(x) + u_2(x) + \dots + u_n(x) + \dots = \sum_{i=1}^{\infty} u_i(x).$$

Это выражение называется *функциональным рядом*. При различных значениях  $x$  из функционального ряда получаются различные числовые ряды

$$a_1 + a_2 + \dots + a_n + \dots = \sum_{i=1}^{\infty} a_i.$$

Числовой ряд может быть сходящимся или расходящимся. Совокупность значений  $x$ , при которой функциональный ряд сходится, называется его областью сходимости.

Числовой ряд называется сходящимся, если сумма  $n$  первых его членов  $S_n = a_1 + a_2 + \dots + a_n$  при  $n \rightarrow \infty$  имеет предел, в противном случае ряд называется расходящимся. Ряд может сходиться лишь при условии, что общий член ряда  $a_n$  при неограниченном увеличении его номера стремится к нулю:  $\lim_{n \rightarrow \infty} a_n = 0$ . Это необходимый признак сходимости для всякого ряда.

В случае бесконечной суммы будем вычислять ее с заданной точностью  $e$ . Считается, что требуемая точность достигается, если вычислена сумма нескольких первых слагаемых и очередное слагаемое оказалось по модулю меньше чем  $e$ .

### Пример 1

---

Написать программу для подсчета суммы  $\sum_{i=1}^{\infty} \frac{(-1)^i}{i!}$  с заданной точностью  $e$  ( $e > 0$ ).

*Указание по решению задачи.* Рассмотрим, что представляет собой заданный ряд:

$$\sum_{i=1}^{\infty} \frac{(-1)^i}{i!} = -\frac{1}{1} + \frac{1}{2} - \frac{1}{6} + \frac{1}{24} - \frac{1}{120} + \dots + \frac{1}{\infty}.$$

Общий член ряда с увеличением значения  $i$  стремится к нулю, следовательно, данную сумму будем вычислять с определенной точностью  $e$ . Заметим также, что последовательность слагаемых можно выразить с помощью рекуррентного соотношения  $a_1 = -1$ ,  $a_i = \frac{-a_{i-1}}{i}$ , а всю сумму — с помощью рекуррентного соотношения  $S_0 = 0$ ,  $S_n = S_{n-1} + a_n$ . (Данные рекуррентные соотношения выведите самостоятельно.)

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int n, i = 1;
    float e, s = 0, a = -1;
    cout << "e = ";
    cin >> e;
    while (fabs(a) >= e) // до тех пор, пока
                           // очередное слагаемое больше e
    {
        s += a; // добавляем его к сумме
        i++; a /= -i; // вычисляем номер очередного слагаемого
                       // и его значение
    }
    cout << "s = " << s << endl;
    return 0;
}
```

Рассмотрим подробно, как выполняется цикл при  $\epsilon = 0.01$ .

Номер шага	Значения переменных до выполнения цикла			Условие $abs(a) \geq \epsilon$	Значения переменных после выполнения цикла		
	<i>i</i>	<i>a</i>	<i>s</i>		<i>i</i>	<i>a</i>	<i>s</i>
1	1	-1	0	TRUE	2	0.5	-1
2	2	0.5	-1	TRUE	3	-0.166...	-0.5
3	3	-0.166...	-0.5	TRUE	4	0.046...	-0.666...
4	4	0.046...	-0.666...	TRUE	5	0.0083...	-0.625
5	5	0.0083...	-0.625	FALSE	5	0.0083...	-0.625

После выполнения программы на экран будет выведено следующее сообщение: «Сумма с заданной точностью равна -0.625».

## Пример 2

Вычислить значение функции

$$F(x) = -\frac{1}{(x+1)} + \frac{(x-1)^2}{2(x+1)^2} - \frac{(x-1)^4}{4(x+1)^3} + \frac{(x-1)^6}{8(x+1)^4} - \dots$$

на отрезке  $[a, b]$  с шагом  $h = 0.1$  и точностью  $\epsilon$ . Результат работы программы представить в виде таблицы, которая содержит номер аргумента, значение аргумента, значение функции и количество просуммированных слагаемых.

*Указания по решению задачи.* Разработаем вспомогательную функцию  $Fun(x, \epsilon, n)$ , которая по заданным значениям  $x$  и  $\epsilon$  вычисляет значение функции и количество слагаемых  $n$ , при суммировании которых была достигнута заданная степень точности. Для этого перейдем от развернутой формы записи функции к сокращенной, получим

$$F(x) = \sum_{i=1}^{\infty} \frac{(-1)^i (x-1)^{2i-2}}{2^{i-1} (x+1)^i}$$

и воспользуемся приемом, рассмотренным в примере 5 параграфа 5.1. Получим, что слагаемые данной функции определяются с помощью рекуррентного соотношения  $a_1 = -\frac{1}{(x+1)}$ ,  $a_i = \frac{-a_{i-1}(x-1)^2}{2(x+1)}$ .

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

float fun(float x, float e, int &n) // вспомогательная функция
{
    // определяем начальное значение суммы и первое слагаемое
    float s = 0, a = -1/(x + 1);
    // определяем количество просуммированных слагаемых
    n = 0;
    // пока не достигнута заданная степень точности
```

```

while (fabs(a) >= e)
{
    s += a; // добавляем слагаемое к сумме
    // формируем очередное слагаемое
    a *= -pow(x - 1,2)/(2*x + 2);
    // увеличиваем количество просуммированных слагаемых
    n++;
}
// возвращаем в качестве значения функции значение s
return s;
}

int main() // главная функция
{
    float a, b, e, h, f, x;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "h = ";
    cin >> h;
    cout << "e = ";
    cin >> e;
    int n, i;
    // устанавливаем количество цифр после запятой
    cout << setprecision(3);
    // выводим заголовок таблицы
    cout << "i\t x\t f(x) \t n\n";
    // строим таблицу на отрезке [a, b]
    for (x = a, i = 1; x <= b; x += h, i++)
    {
        // вызываем вспомогательную функцию
        f = fun(x, e, n);
        // выводим полученные данные
        cout << i << "\t" << x << "\t" << f << "\t" << n << endl;
    }
    return 0;
}

```

Результат работы программы для отрезка [1,2],  $h = 0.3$ ,  $e = 0.00001$ :

i	x	f(x)	n
1	1	-0.5	1
2	1.3	-0.426	2
3	1.6	-0.36	4
4	1.9	-0.303	6

## Упражнения

I. Для заданного натурального  $n$  и действительного  $x$  подсчитать следующие суммы:

$$1) S = 1^2 + 2^2 + 3^2 + \dots + n^2;$$

$$2) S = \sqrt{1} + \sqrt{2} + \sqrt{3} + \dots + \sqrt{n};$$

$$3) S = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n};$$

$$4) S = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{n^2};$$

$$5) S = 1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + \dots + \frac{1}{\sqrt{n}};$$

$$6) S = \frac{1}{\sin 1} + \frac{1}{\sin 2} + \dots + \frac{1}{\sin n};$$

$$7) S = 1 + 2 + 2^2 + 2^3 + \dots + 2^n;$$

$$8) S = \cos 1 - \cos 2 + \cos 3 - \dots + (-1)^{n+1} \cos n;$$

$$9) S = 1! + 2! + 3! + \dots + n!;$$

$$10) S = 1 - 3 + 3^2 - 3^3 + \dots + (-1)^n 3^n;$$

$$11) S = 1! - 2! + 3! - \dots + (-1)^{n+1} n!;$$

$$12) S = \sin x + \sin x^2 + \sin x^3 + \dots + \sin x^n;$$

$$13) S = 1 + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{n!};$$

$$14) S = -\frac{1}{2} + \frac{1}{2^2} - \frac{1}{2^3} + \dots + \frac{(-1)^n}{2^n};$$

$$15) S = 1^3 - 2^3 + 3^3 + \dots + (-1)^{n+1} n^3;$$

$$16) S = x + 3x^3 + 5x^5 + 7x^7 + \dots + (2n-1)x^{2n-1};$$

$$17) S = \frac{\cos x}{1} + \frac{\cos^2 x}{2} + \frac{\cos^3 x}{3} + \dots + \frac{\cos^n x}{n};$$

$$18) S = \frac{1}{3^2} - \frac{1}{5^2} + \frac{1}{7^2} - \dots + \frac{(-1)^{n+1}}{(2n+1)^2};$$

$$19) S = \frac{1}{\sin 1} + \frac{1}{\sin 1 + \sin 2} + \dots + \frac{1}{\sin 1 + \sin 2 + \dots + \sin n};$$

$$20) S = \sin x + \sin \sin x + \sin \sin \sin x + \dots + \underbrace{\sin \sin \sin \dots \sin x}_{n \text{ раз}}.$$

**II. Для заданного натурального  $k$  и действительного  $x$  подсчитать следующие выражения:**

$$1) S = \sum_{n=1}^k \frac{x^n}{n};$$

$$2) S = \sum_{n=1}^k \frac{2^n \cdot n!}{n^2};$$

$$3) S = \sum_{n=1}^k \frac{(-1)^{n+1}}{n^2};$$

$$4) S = \sum_{n=1}^k \frac{x^{2(n-1)}}{[2+4(n-1)]^2};$$

$$5) S = \sum_{n=1}^k \frac{(-1)^{n+1} x^{2n-1}}{(2n-1)!};$$

$$6) S = \sum_{n=1}^k \frac{1}{n \cdot n!};$$

$$7) S = \sum_{n=0}^k \frac{(-1)^{n+1} x^{2n+1}}{(2n+1)!};$$

$$8) S = \sum_{n=1}^k \frac{(-1)^{n-1} x^n}{(2n)!};$$

$$9) S = \sum_{n=1}^k \frac{(-1)^{n-1} x^{2n}}{(2n)!};$$

$$10) S = \sum_{n=1}^k \frac{(-1)^n x^n}{2^n 7^n};$$

$$11) P = \prod_{n=1}^k \left( 1 + \frac{x^n}{n^2} \right);$$

$$12) P = \prod_{n=1}^k \left[ 1 + \frac{x^{2n+1}}{n(n+1)} \right];$$

$$13) P = \prod_{n=1}^k \left( 1 - \frac{x^n}{n!} \right);$$

$$14) P = \prod_{n=1}^k \left( 1 + \frac{(-1)^n x^{2n}}{n^3} \right);$$

$$15) P = \prod_{n=1}^k \left( 1 + \frac{(-1)^{n+1} x^n}{n!} \right);$$

$$16) P = \prod_{n=1}^k \left[ 1 + \frac{x^{2n}}{n(n+4)} \right];$$

$$17) P = \prod_{n=1}^k \left( 1 + \frac{(-1)^n x^{2n+1}}{n^3 + n^2} \right);$$

$$18) P = \prod_{n=1}^k \left( 1 + \frac{x^n}{2n!} \right);$$

$$19) P = \prod_{n=2}^{\infty} \left( 1 + \frac{(-1)^n x^{2n-1}}{n^3 - 1} \right);$$

$$20) P = \prod_{n=0}^k \left[ 1 + \frac{(-1)^{n-1} x^{2n}}{(n+2)(n+1)} \right].$$

**III.** Вычислить бесконечную сумму ряда с заданной точностью  $e$  ( $e > 0$ ):

$$1) \sum_{i=1}^{\infty} \frac{1}{i^2};$$

$$2) \sum_{i=1}^{\infty} \frac{1}{(i+1)^3};$$

$$3) \sum_{i=2}^{\infty} \frac{(-1)^i}{i^2 - 1};$$

$$4) \sum_{i=1}^{\infty} \frac{1}{i(i+1)};$$

$$5) \sum_{i=2}^{\infty} \frac{5}{(i+1)(i-1)};$$

$$6) \sum_{i=1}^{\infty} \frac{(-2)^{i+1}}{i(2i+1)};$$

$$7) \sum_{i=1}^{\infty} \frac{2}{i!};$$

$$8) \sum_{i=1}^{\infty} \frac{1}{(2i)!};$$

$$9) \sum \frac{(-1)^i}{(2i-1)!};$$

$$10) \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{2i!};$$

$$11) \sum_{i=1}^{\infty} \frac{(-1)^{2i}}{i(i+1)(i+2)};$$

$$12) \sum_{i=3}^{\infty} \frac{(-1)^{2i-1}}{i(i-1)(i-2)};$$

$$13) \sum_{i=1}^{\infty} \frac{(-3)^{2i}}{3i!};$$

$$14) \sum_{i=1}^{\infty} \frac{(-5)^{2i-1}}{5(2i-1)!};$$

$$15) \sum_{i=1}^{\infty} \frac{1}{2^i};$$

$$16) \sum_{i=1}^{\infty} \frac{1}{3^i + 4^i};$$

$$17) \sum_{i=1}^{\infty} \frac{1}{5^i + 4^{i+1}};$$

$$18) \sum_{i=1}^{\infty} \frac{(-1)^i}{2^{2i}};$$

$$19) \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{3^{2i-1}};$$

$$20) \sum_{i=1}^{\infty} \frac{1}{\sqrt{3^i}}.$$

**IV.** Вычислить и вывести на экран значение функции  $F(x)$  на отрезке  $[a, b]$  с шагом  $h = 0.1$  и точностью  $\varepsilon$ . Результат работы программы представить в виде следующей таблицы.

№ п/п	Значение $x$	Значение функции $F(x)$	Количество просуммирован- ных слагаемых $n$
1			
2			
...			

При решении задачи использовать вспомогательную функцию.

$$1. F(x) = 1 + \frac{x^2}{4} + \frac{x^3}{4^2} + \frac{x^4}{4^3} + \frac{x^5}{4^4} + \dots, x \in [0.1; 0.9].$$

$$2. F(x) = 1 - \frac{x}{2 \cdot 7} + \frac{x^2}{4 \cdot 14} - \frac{x^3}{8 \cdot 21} + \frac{x^4}{16 \cdot 28} - \dots, x \in [0; 0.9].$$

$$3. F(x) = 1 - \frac{x^2}{1 \cdot 3 \cdot 4} + \frac{x^4}{2 \cdot 4 \cdot 5} - \frac{x^6}{3 \cdot 5 \cdot 6} + \frac{x^8}{4 \cdot 6 \cdot 7} - \dots, x \in [0.2; 0.7].$$

$$4. F(x) = 1 + \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 2^2} + \frac{x^7}{7 \cdot 2^3} + \dots, x \in [0; 0.99].$$

$$5. F(x) = 1 + \frac{x}{1 \cdot 4} - \frac{x^2}{2 \cdot 5} + \frac{x^3}{3 \cdot 6} - \frac{x^4}{4 \cdot 7} + \dots, x \in [0.1; 0.9].$$

$$6. F(x) = 1 - \frac{x^3}{3 \cdot 4^2} + \frac{x^5}{4 \cdot 5^2} - \frac{x^7}{5 \cdot 6^2} + \dots, x \in [0; 0.8].$$

$$7. F(x) = 1 + \frac{x}{3} + \left(\frac{x}{3}\right)^2 + \dots, x \in [0; 0.9].$$

$$8. F(x) = 1 + \frac{x^2}{2 \cdot 4} + \frac{x^3}{4 \cdot 6} + \frac{x^4}{6 \cdot 8} + \dots, x \in [0.2; 0.6].$$

$$9. F(x) = 1 + \frac{x}{1 \cdot 3} + \frac{x^2}{1 \cdot 3 \cdot 5} + \frac{x^3}{1 \cdot 3 \cdot 5 \cdot 7} + \dots, x \in [0.05; 0.95].$$

$$10. F(x) = -\frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \frac{1}{7x^7} - \dots, x \in [-3; -2].$$

$$11. F(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots, x \in [0; 1].$$

$$12. F(x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \dots, x \in [2; 3].$$

$$13. F(x) = 1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!} + \frac{x^8}{4!} + \dots, x \in [-1; 0].$$

$$14. F(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots, x \in [1; 2].$$

$$15. F(x) = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \frac{x^8}{9!} - \dots, x \in [0; 1].$$

$$16. F(x) = -\frac{x}{2!} + \frac{x^3}{4!} - \frac{x^5}{6!} + \frac{x^7}{8!} - \dots, x \in [-1; 0].$$

$$17. F(x) = 2 \left[ \frac{x-1}{x+1} + \frac{(x-1)^3}{3(x+1)^3} + \frac{(x-1)^5}{5(x+1)^5} + \dots \right], x \in [1; 2].$$

$$18. F(x) = \frac{x-1}{x} + \frac{(x-1)^2}{2x^2} + \frac{(x-1)^3}{3x^3} + \dots, x \in [1; 2].$$

$$19. F(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \dots, x \in [0.5; 1.5].$$

$$20. F(x) = \frac{\pi}{2} - \left( x + \frac{x^3}{2 \cdot 3} + \frac{3x^5}{2 \cdot 4 \cdot 5} + \frac{3 \cdot 5x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \frac{3 \cdot 5 \cdot 7x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} + \dots \right), x \in [-0.9; 0.9].$$

# Глава 6

## МАССИВЫ

### 6.1. Указатели

Когда компилятор обрабатывает оператор определения переменной, например `int a = 50;`, то он выделяет память в соответствии с типом `int` и записывает в нее значение 50. Все обращения в программе к переменной по ее имени заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Программист может определить собственные переменные для хранения адресов области памяти. Такие переменные называются **указателями**. В C++ различают три вида указателей — указатели на объект, на функцию и на `void`, которые отличаются друг от друга свойствами и допустимыми операциями. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим базовым типом.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа (простого или составного). Простейшее объявление указателя на объект имеет следующий вид:

```
<базовый тип> [<модификатор>] * <имя указателя>;
```

где **базовый тип** — имя типа переменной, адрес которой будет содержать переменная указатель; тип может быть любым, кроме ссылки (см. следующую главу) и битового поля (см. справочники по C++); **модификатор** необязателен и может иметь значение: *near*, *far* или *huge* (см. справочники по C++).

#### Замечание

По умолчанию устанавливается модификатор *near*. Нам этого будет достаточно для решения задач, поэтому при описании указателей модификатор явным образом указывать не будем.

Указатель может быть переменной или константой, указывать на переменную или константу, а также быть указателем на указатель. Например:

```
int i; // целочисленная переменная
const int j = 10; // целочисленная константа
int *a; // указатель на целочисленное значение
```

```

int **x;           // указатель на указатель на целочисленное
                  // значение
const int *b;     // указатель на целочисленную константу
int * const c = &i; // указатель-константа на целочисленную
                  // переменную
const int *const d = &j; // указатель-константа на целую переменную

```

Как видно из примера, модификатор *const*, находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а *const* слева от звездочки задает постоянство значения, на которое он указывает.

Величины типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни. Память под указатели выделяется в сегменте данных или в стеке (в зависимости от места описания и спецификатора класса памяти), а область памяти, связанная с указателем, обычно выделяется в динамической памяти (куче) (см. параграф 2.5).

Указатель на функцию содержит адрес в сегменте кода, по которому передается управление при вызове функции. Указатели на функцию используются для косвенного вызова функции (не через ее имя, а через переменную, хранящую ее адрес), а также для передачи функции в другую функцию в качестве параметра.

#### Замечание

---

Более подробно данный вид указателей будет рассмотрен далее.

Указатель *типа void* применяется в тех случаях, когда конкретный тип объекта, адрес которого нужно хранить, не определен. Указателю на **void** можно присвоить значение указателя любого типа, а также сравнить его с любым указателем, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

Перед использованием любого указателя надо выполнить его *инициализацию*, т.е. произвести присваивание начального значения. Существуют следующие способы инициализации указателя:

- 1) присваивание указателю адреса существующего объекта:

- с помощью операции получения адреса:

```

int a = 50; // целая переменная
int *x = &a; // указателю присваивается адрес целой переменной a
int *y (&a); // указателю присваивается адрес целой переменной a

```

- с помощью значения другого инициализированного указателя:

```
int *z = x; // указателю присваивается адрес, хранящийся в x
```

- с помощью имени массива или функции (рассмотрим позже);

- 2) присваивание указателю адреса области памяти в явном виде:

```
int *p = (int *) 0xB8000000;
```

где 0xB8000000 — шестнадцатеричная константа, (*int* \*) — операция явного приведения типа к типу указатель на целочисленное значение;  
3) присваивание пустого значения:

```
int *x = NULL;
int *y = 0;
```

где NULL — стандартная константа, определенная как указатель, равный 0;

4) выделение участка динамической памяти и присваивание ее адреса указателю:

```
int *a = new int;      // 1
int *b = new int (50); // 2
```

В строке 1 операция **new** выполняет выделение достаточного для размещения величины типа **int** участка динамической памяти и записывает адрес начала этого участка в переменную *a*. Память под переменную *a* выделяется на этапе компиляции. В строке 2, кроме действий, описанных выше, производится инициализация выделенной динамической памяти значением 50.

Освобождение памяти, выделенной с помощью операции **new**, должно выполняться с помощью операции **delete**. При этом переменная-указатель сохраняется и может инициализироваться повторно. Пример использования операции **delete**:

```
delete a; delete []b;
```

С указателями можно выполнять операции разадресации или косвенного обращения к объекту, получения адреса, присваивания, сложения с константой, вычитания, инкремент, декремент, сравнения и приведения типов.

*Операция разадресации* (\*) предназначена для доступа к значению, адрес которого хранится в указателе. Эту операцию можно использовать как для получения значения, так и для его изменения. Рассмотрим пример.

```
#include <iostream>
using namespace std;
int main()
{
    // инициализация указателя на целочисленное значение
    int *a = new int(50);
    // переменной b присваивается значение,
    // хранящееся по адресу указателя a
    int b = *a;
    cout << "adress \t *a\t b\n";
    /* выводим: адрес, хранящийся в указателе a; значение, хранящееся
       по адресу указателя a; значение переменной b */
    cout << a << "\t" << *a << "\t" << b << endl;
    *a = 100; // изменяем значение, хранящееся по адресу указателя a
```

```
    cout << a << "\t" << *a << "\t" << b << endl;
    return 0;
}
```

Результат работы программы:

```
adress      *a      b
00355900    50      50
00355900    100     50
```

### Замечания

1. При запуске данной программы на вашем компьютере будет выведен другой адрес, записанный в указателе *a*. Это связано с текущим состоянием динамической памяти.
2. При попытке выполнить команду *a = 100*; возникнет ошибка, так как *a* — переменная-указатель, в ней может храниться только адрес ячейки оперативной памяти, но не целочисленное значение.
3. При попытке выполнить команду *\*b = 100*; также возникнет ошибка, так как операция разадресации может применяться только к указателям, а не к обычным переменным.

*Операция получения адреса (&)* применима к величинам, имеющим имя и размещенным в оперативной памяти. Рассмотрим пример.

```
#include <iostream>
using namespace std;
int main()
{
    int b = 50;
    int *a = &b; // В указатель a записали адрес переменной b
    cout << "adress \t *a\t b\n";
    cout << a << "\t" << *a << "\t" << b << endl;
    b += 10; // 1
    cout << a << "\t" << *a << "\t" << b << endl;
    *a = 100; // 2
    cout << a << "\t" << *a << "\t" << b << endl;
    return 0;
}
```

Результат работы программы:

```
adress      *a      b
0012FF60    50      50
0012FF60    60      60
0012FF60    100     100
```

### Замечание

После того как в указатель *a* был записан адрес переменной *b*, мы получили доступ к одной и той же области памяти через имя обычной переменной и через указатель. Поэтому, изменяя в строке 1 значение переменной *b*, мы

фактически изменяем значение по адресу, записанному в указателе *a*, и наоборот (см. строку 2).

---

*Арифметические операции с указателями* (сложение с константой, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл при работе со структурами данных, последовательно размещенными в памяти, например с массивами.

Более подробно эти операции будут рассмотрены при изучении массивов. Однако хотелось обратить внимание на следующий пример:

```
#include <iostream>
using namespace std;
int main()
{
    int *a = new int(50);
    cout << "adress \t *a\n";
    cout << a << "\t" << *a << endl;
    // 1 увеличивается на 1 значение, хранящееся по адресу указателя a
    (*a)++;
    cout << a << "\t" << *a << endl;
    // 2 значение указателя изменяется на величину sizeof (int)
    *(a++);
    cout << a << "\t" << *a << endl;
    // 3 с учетом приоритета операций * и ++ аналог строки 2
    *a++;
    cout << a << "\t" << *a << endl;
    return 0;
}
```

Результат работы программы:

```
adress      *a
00355900    50
00355900    51
00355904    -31686019
00355908    393224
```

#### Замечание

В строке 2 значение указателя изменилось на величину *sizeof (int)* — в нашем случае на 4 байта, в результате чего от адреса 00355900 мы перешли к адресу 00355904. Так как данные по новому адресу нами не были инициализированы, то на экран вывелоось случайное число. Аналогичным образом была выполнена строка 3.

---

*Операция присваивания* используется для изменения значения указателя или значения, которое хранится по адресу, записанному в указателе. Использование данной операции мы рассмотрели в ходе изучения других операций.

## 6.2. Ссылки

**Ссылка** представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который разыменовывается неявным образом.

Формат объявления ссылки:

<базовый тип> & <имя ссылки>

Например:

```
int a;      // целочисленная переменная
int &b = a; // ссылка на целочисленную переменную a
```

Рассмотрим следующий пример:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 50; // целочисленная переменная a
    int &b = a; // ссылка b - альтернативное имя для переменной a
    cout << "a\t b\n";
    cout << a << "\t" << b << endl;
    a++; // 1
    cout << a << "\t" << b << endl;
    b++; // 2
    cout << a << "\t" << b << endl;
    return 0;
}
```

Результат работы программы:

```
a      b
50    50
51    51
52    52
```

### Замечание

Переменная *a* и ссылка *b* обращаются к одному и тому же участку памяти, поэтому при изменении значения *a* изменяется значение *b*, и наоборот. Однако, в отличие от указателей, для обращения к значению по ссылке (см. строку 2) не надо использовать операцию разадресации, так как она выполняется неявным образом.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых значений. В отличие от указателей они не занимают дополнительного пространства в памяти и являются просто другим именем величины. Однако при работе со ссылками нужно помнить следующие правила:

- 1) тип ссылки должен совпадать с типом величины, на которую она ссылается;
- 2) переменная-ссылка должна быть явно инициализирована при описании, кроме случаев, когда она является параметром функции, имеет спецификатор `extern` или ссылается на поле данных класса;
- 3) повторная инициализация ссылки невозможна;
- 4) не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки ссылок.

## 6.3. Одномерные массивы

---

**Одномерный массив** — это фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент имеет свой номер.

---

Нумерация элементов массива в C++ начинается с нулевого элемента, т.е. если массив состоит из 10 элементов, то его элементы будут иметь следующие номера: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Элементы массива могут быть любого типа, в том числе и структурированного (кроме файлового).

Между указателями и массивами существует взаимосвязь: любое действие над элементами массивов, которое достигается индексированием, может быть выполнено и с помощью указателей. Вариант программы с указателями будет работать быстрее, но для понимания он сложнее. Рассмотрим различные варианты реализации одномерных массивов.

**Статические одномерные массивы.** Формат объявления статического массива:

`<базовый тип> <имя массива>[<размер массива>] = {<список инициализации>}`

### Замечание

---

В данном случае [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.

---

Одновременно с объявлением массива может проводиться его инициализация, например:

```
// массив из 10 элементов с плавающей точкой, не инициализирован
float v[10];
// массив из 3 элементов целого типа,
// инициализирован списком чисел 1, 2, 3
int a[] = {1,2,3};
// массив из 5 элементов целого типа,
// инициализирован списком чисел 1, 2, 3, 0, 0
int b[5] = {1,2,3};
```

Память под статический массив выделяется на этапе объявления массива в стеке или сегменте данных (в зависимости от спецификатора класса памяти, указанного явно или используемого по умолчанию).

Для обращения к элементу массива указывается имя массива, а затем в квадратных скобках — индекс (номер). В примере, рассмотренном выше, для обращения к элементу массива *a* с номером 2 нужно записать *a[2]*, к элементу с номером ноль — *a[0]*. Индексом может быть любое целое выражение, образуемое целыми переменными и целыми константами.

Ввод и вывод массивов производится поэлементно. В следующем фрагменте программы производится потоковый ввод и вывод 10 элементов целочисленного массива:

```
int b[10]; // объявление массива
for (int i = 0; i < 10; i++) // ввод элементов массива
{
    cout << "b[" << i << "] = ";
    cin >> b[i];
}
...
for (int i = 0; i < 10; i++) // вывод элементов массива на экран
    cout << "b[" << i << "] = " << b[i] << "\n";
```

### Замечание

При работе со статическими одномерными массивами можно использовать указатели.

Например:

```
int b[10]; // объявление массива b
// определяем указатель на тип int и устанавливаем
// его на нулевой элемент массива b
int *p = &b[0];
for (int i = 0; i < 10; i++) // ввод элементов массива
{
    cout << "b[" << i << "] = ";
    // введенное значение помещаем по адресу указателя p и сдвигаем
    // указатель на следующий элемент массива
    cin >> *p++;
}
...
// повторно устанавливаем указатель на нулевой элемент массива b
p = &b[0];
for (int i = 0; i < 10; i++) // вывод элементов массива
    // выводим на экран значение, хранящееся по адресу
    // указателя p, и сдвигаем указатель на следующий элемент
    // массива
    cout << "b[" << i << "] = " << *p++ << endl;
```

Команда *\*p++* с учетом приоритетов операций эквивалентна команде *(\*p)++*. При этом вначале будет произведено обращение к значению, которое хранится по адресу, записанному в указатель *p*, а затем этот адрес изменится на столько байт, сколько требуется для размещения в памяти базового типа указателя. Для одномерного массива, в кото-

ром элементы располагаются последовательно друг за другом, произойдет переход к адресу следующего элемента массива.

**Динамические одномерные массивы** отличаются от статических тем, что:

- 1) имя массива является указателем-переменной;
- 2) память под элементы массива выделяется в куче с помощью специальной команды *new*.

Формат объявления динамического массива:

```
<базовый тип> * <имя массива> = new <базовый тип> [<размерность массива>];
```

### Замечание

В данном случае [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.

Например:

```
// вещественный массив из 10 элементов, не инициализирован
float *v = new float [10];
// массив из 3 элементов целого типа, не инициализирован
int *a = new int [3];
```

Обращаться к элементам массива можно через индексы, как и к элементам статических массивов, а можно и через указатели. Например, чтобы обратиться к нулевому элементу массива, можно написать *a[0]* или *\*a*, а для обращения к элементу массива *a* с номером *i* нужно написать *a[i]* или *\*(a+i)*.

Рассмотрим более подробно обращение к элементам массива через указатели. Указатель *a* хранит адрес нулевого элемента массива, поэтому, когда мы выполняем операцию *\*a*, мы обращаемся к значению, которое хранится по данному адресу. Выражение *\*(a+2)* говорит о том, что вначале мы сдвинемся по отношению к нулевому элементу массива *a* на столько байт, сколько занимают два элемента, а затем обратимся к значению, которое там хранится.

Рассмотренные способы обращения к элементам массива эквивалентны друг другу. Однако первый способ более понятен, а во втором случае программа будет выполняться быстрее, так как компилятор автоматически переходит от индексного способа обращения к массиву к обращению через указатели.

Ввод и вывод массивов, так же как и в случае со статическими массивами, производится поэлементно. В следующем фрагменте программы производится потоковой ввод и вывод 10 элементов целочисленного массива:

```
// объявили динамический массив для хранения 10 целых чисел
int *b = new int [10];
// ввод элементов массива
```

```

for (int i = 0; i < 10; i++)
{
    cout << "b[" << i << "] = ";
    // Введенное значение записываем в i-й элемент массива b
    cin >> *(b + i);
}
...
for (int i = 0; i < 10; i++)
    // выводим на экран значение i-го элемента массива b
cout << "b[" << i << "] = " << *(b+i) << endl;

```

Освободить память, выделенную под динамический массив, можно с помощью операции **delete**. Например, освобождение памяти, выделенной под элементы массива *b* из предыдущего примера, производится следующим образом:

```
delete [] b;
```

### Замечание

Если в данной записи пропустить квадратные скобки, то ошибки не будет, однако освобождена будет память, выделенная под нулевой элемент массива, а с остальными элементами массива «связь» будет потеряна и они не будут доступны для дальнейших операций. Такие ячейки памяти называются мусором. Кроме того, операция **delete** освобождает память, выделенную в куче под элементы массива, а сам указатель *b* при этом сохраняется и может повторно использоваться для определения другого массива.

**Передача одномерных массивов в качестве параметров.** При использовании в качестве параметра одномерного массива в функцию передается указатель на его нулевой элемент. Напомним, что указателем на нулевой элемент является имя одномерного массива. Операцию раз-адресации внутри функции при обращении к элементам массива выполнять не надо, так как запись вида *a[i]* автоматически заменяется компилятором на выражение *\*(a+i)*.

При передаче массива в качестве параметра функции информация о количестве элементов теряется, поэтому размерность массива следует передавать в качестве отдельного параметра. Рассмотрим несколько примеров.

1. Передача статического массива:

```

#include <iostream>
using namespace std;
/* В функцию в качестве параметров передаются статический массив mas
и его размерность n */
void print (int mas[], int n)
{
    for (int i = 0; i < n; i++) cout << mas[i] << "\t";
    cout << endl;
}

```

```

int main()
{
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    print(a, 10);
    cout << endl;
    print (a, 7);
    print(a, 13);
    return 0;
}

```

Результат работы программы:

```

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7 8 9 -858993406 1245112 4269558

```

### Замечание

---

При втором вызове функции *print* в качестве параметра *n* мы передали значение, меньшее реального размера массива, поэтому мы смогли распечатать не весь массив, а его часть. Если же в функцию в качестве параметра *n* передать значение больше реального размера массива, то на экран будут выведены случайные числа. Это объясняется тем, что будут задействованы участки памяти, которые не закреплены за массивом. Подумайте, чем опасен для программы выход за границы массива.

---

## 2. Передача динамического массива

```

#include <iostream>
using namespace std;
/* В функцию в качестве параметров передаются динамический массив mas
и его размерность n */
void print (int *mas, int n)
{
    for (int i = 0; i < n; i++)
        cout << mas[i] << "\t";
}
int main()
{
    int n = 10;
    int *a = new int [n];
    for (int i = 0; i < n; i++)
        a[i] = i * i;
    print(a, 10);
    return 0;
}

```

Результат работы программы:

```

0 1 4 9 16 25 36 49 64 81

```

Поскольку массив всегда передается в функцию по адресу, то все изменения, внесенные в формальный параметр массива, отразятся

на фактическом параметре массива. Если необходимо запретить изменение массива в функции, то его нужно передавать как параметр-константу. Например:

```
int sum (const int *mas, const int n)
{ ... }
```

**Массив как возвращаемое значение функции.** Массив может возвращаться в качестве значения функции только как указатель:

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

int * creat(int n)
{
    int *mas = new int [n];
    for (int i = 0; i<n; i++)
        mas[i] = i*i;
    return mas;
}

void print (int *mas, int n)
{
    for (int i = 0; i < n; i++)
        cout << mas[i] << "\t";
}

int main()
{
    int n = 5;
    int *a = creat(n);
    print(a, n);
    return 0;
}
```

Результат работы программы:

```
0  1  4  9  16
```

#### Замечание

В общем случае нельзя возвращать из функции указатель на локальную переменную, поскольку память, выделенная под локальную переменную, освобождается после выхода из функции. Например:

```
int *f()
{
    int a = 5;
    return &a; // нельзя!!!
}
```

Однако в случае с динамическим массивом *mas*, который является указателем на область данных в куче, такое действие возможно. Команда *return mas* возвратит в качестве значения функции адрес в куче, начиная с которого

последовательно хранятся элементы массива. После этого локальная переменная *mas* перестанет существовать, и освободится соответствующая ей область стека (см. параграф 2.5). Но куча при этом не освобождается, поэтому элементы массива будут нам доступны.

---

## 6.4. Примеры использования одномерных массивов

Одномерные массивы удобно использовать тогда, когда данные можно представить в виде некоторой последовательности, элементы которой проиндексированы. Напомним, что индексация элементов в одномерном массиве в C++ начинается с нуля. При решении задач мы будем использовать статические массивы и обращаться к элементам через индексы. Однако мы советуем вам попробовать модифицировать данные задачи для работы с динамическими массивами.

1. Дан массив из  $n$  целых чисел ( $n \leq 20$ ). Написать программу, которая заменяет в данном массиве все отрицательные элементы нулями.

```
#include <iostream>
using namespace std;
int main()
{
    int a[20]; // объявляем статический массив,
                // задав максимальную размерность
    int n;
    cout << "n = ";
    cin >> n; // ввели количество элементов массива
    for (int i = 0; i < n; ++i) // ввод и обработка данных
    {
        cout << "a[" << i << "] = ";
        cin >> a[i]; // ввод очередного элемента
        if (a[i] < 0) // если i-й элемент массива отрицательный,
            a[i] = 0; // заменяем его нулем
    }
    for (int i = 0; i < n; ++i) // вывод массива на экран
        cout << a[i] << "\t";
    return 0;
}
```

Результат работы программы:

```
Исходные данные
2 -4 1 2 -2 0 23 -12 1 -1
Измененные данные
2 0 1 2 0 0 23 0 1 0
```

2. Дан массив из  $n$  действительных чисел ( $n \leq 100$ ). Написать программу для подсчета суммы этих чисел.

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

```

int main()
{
    float a[100];
    int n;
    cout << "n = ";
    cin >> n;
    float s = 0;
    for (int i = 0; i < n; ++i)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
        s += a[i]; // добавление значения элемента массива к сумме
    }
    cout << "s = " << s << endl;
    return 0;
}

```

Результат работы программы:

n	Исходные данные	Ответ
5	2.3 0 2.5 1.7 -1.5	S = 5

#### Замечание

---

Обратите внимание на то, что при подсчете суммы мы использовали уже известный нам из гл. 5 прием накопления суммы  $s += a[i]$ .

---

**3.** Дан массив из  $n$  целых чисел ( $n \leq 50$ ). Написать программу для подсчета среднего арифметического четных значений данного массива.

```

#include <iostream>
using namespace std;
int main()
{
    int a[50];
    int n, k = 0;
    cout << "n = ";
    cin >> n;
    float s = 0;
    for (int i = 0; i < n; ++i)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
        if (!(a[i] % 2)) // если остаток при делении элемента на 2 равен 0
        {
            s += a[i]; // то элемент четный - добавить его к сумме
            ++k; // и увеличить количество четных элементов на 1
        }
    }
    // если k не нулевое, то четные числа в последовательности есть
    // и можно вычислить их среднее арифметическое значение
    if (k)
        cout << "sr = " << s/k << endl;
}

```

```

    else cout << " четных чисел в последовательности нет " << endl;
return 0;
}

```

Результат работы программы:

Исходные данные	Ответ
1 3 7 -41 9	четных чисел в последовательности нет
2 4 6 4	sr = 4.00

### Замечание

---

Выражение  $a[i]\%2$  будет давать 0, если  $a[i]$  — четное число. В C++ 0 трактуется как ложь, поэтому в операторе **if** мы ставим операцию логического отрицания (!) перед этим выражением.

---

4. Дан массив из  $n$  целых чисел ( $n \leq 30$ ). Написать программу, которая определяет наименьший элемент в массиве и его порядковый номер.

```

#include <iostream>
using namespace std;
int main()
{
    int a[30];
    int n; cout << "n = ";
    cin >> n;
    for (int i = 0; i < n; ++i)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }
    // в качестве наименьшего значения полагаем нулевой элемент массива
    int min = a[0];
    int nmin = 0; // соответственно его порядковый номер равен 0
    // перебираем все элементы массива с первого по последний
    for (int i = 1; i < n; ++i)
        // если очередной элемент окажется меньше значения min, то
        if (a[i] < min)
        {
            // в качестве нового наименьшего значения запоминаем
            // значение текущего элемента массива i,
            // соответственно, запоминаем его номер
            min = a[i];
            nmin = i;
        }
    cout << "min = " << min << "\t nmin = " << nmin << endl;
    return 0;
}

```

Результат работы программы:

Исходные данные	Ответ
1 3 7 -41 9	min = 41 nmin = 4

5. Дан массив из  $n$  действительных чисел ( $n \leq 20$ ). Написать программу, которая меняет местами в этом массиве наибольший и наименьший элемент местами (считается, что в последовательности только один наибольший и один наименьший элементы).

```
#include <iostream>
using namespace std;
int main()
{
    float a[20];
    int n;
    cout << "n = ";
    cin >> n;
    for (int i = 0; i < n; ++i)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }
    // первоначально полагаем элемент с номером 0 минимальным
    // и максимальным и запоминаем его номер
    float min = a[0], max = a[0];
    int nmin = 0, nmax = 0;
    // поиск наибольшего и наименьшего значения в массиве и их номеров
    for (int i = 1; i < n; ++i)
    {
        if (a[i] < min)
        {
            min = a[i];
            nmin = i;
        }
        if (a[i] > max)
        {
            max = a[i];
            nmax = i;
        }
    }
    // в позицию наименьшего элемента записываем значение наибольшего
    a[nmax] = min;
    // в позицию наибольшего элемента записываем значение наименьшего
    a[nmin] = max;
    for (int i = 0; i < n; ++i) // выводим измененный массив на экран
        cout << a[i] << "\t";
    return 0;
}
```

Результат работы программы:

n	Исходные данные	Измененные данные
4	1.1 3.4 -41.2 9.9	1.1 3.4 9.9 -41.2

6. Дан массив из  $n$  действительных чисел ( $n \leq 20$ ). Написать программу, которая подсчитывает количество пар соседних элементов массива, для которых предыдущий элемент равен следующему.

```

#include <iostream>
using namespace std;
int main()
{
    float a [20];
    int n, k = 0;
    cout << "n = ";
    cin >> n;
    for (int i = 0; i < n; ++i)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }
    for (int i = 0; i < n - 1; ++i)
        if (a[i] == a[i + 1]) // если соседние элементы равны,
            ++k; // то количество искомых пар увеличиваем на 1
    cout << "k = " << k << endl;
    return 0;
}

```

Результат работы программы:

n	Исходные данные	Ответ
6	1.1 3.4 -41.2 9.9 3.1 -3.7	k = 0
6	1.1 1.1 -41.2 -3.7 -3.7 -3.7	k = 3

#### Замечание

Обратите внимание на то, что в последнем цикле параметр  $i$  принимает значения от 0 до  $n - 2$ , а не до  $n - 1$ . Это связано с тем, что для  $i = n - 2$  существует пара с номерами  $(n - 2, n - 1)$ , а для  $i = n - 1$  пары с номерами  $(n - 1, n)$  не существует, так как в массиве всего  $n$  элементов и последний элемент имеет номер  $n - 1$ .

## 6.5. Двумерные массивы

*Двумерные* массивы (матрицы, таблицы) представляют собой фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент определяется номером строки и номером столбца, на пересечении которых он находится. Нумерация строк и столбцов начинается с нулевого номера. Поэтому если массив содержит три строки и четыре столбца, то строки нумеруются: 0, 1, 2; а столбцы: 0, 1, 2, 3. В C++ двумерный массив реализуется как одномерный, каждый элемент которого также массив. При этом массивы бывают статические и динамические.

**Статические двумерные массивы.** Формат объявления статического двумерного массива:

<базовый тип> <имя массива>[число строк][число столбцов]

## Замечания

1. В данном случае квадратные скобки [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.
2. Память под статический массив выделяется на этапе объявления массива в стеке или сегменте данных (в зависимости от спецификатора класса памяти, указанного явно или используемого по умолчанию).

Одновременно с объявлением массива может проводиться его инициализация, например:

```
// вещественный массив из 4 строк и 3 столбцов,
// массив не инициализирован
float a[4][3];

/*целочисленный массив из 3 строк и 2 столбцов, в нулевой строке
записаны единицы, в первой строке - двойки, во второй строке -
тройки */
int b[3][2] = {1,1, 2, 2, 3, 3};

/* аналог массива b, но так как количество строк и столбцов
не указано явным образом, то содержимое каждой строки помещается
в дополнительные фигурные скобки*/
int c[][] = {{1,1}, {2, 2}, {3, 3}};
```

Для обращения к элементу двумерного массива нужно указать имя массива, номер строки и номер столбца (каждый в отдельной паре квадратных скобок), на пересечении которых находится данный элемент массива. Например, обратиться к элементу массива *a*, расположенному на пересечении строки с номером 0 и столбца с номером 3, можно так: *a[0][3]*.

Ввод и вывод двумерного массива осуществляется поэлементно. В следующем фрагменте программы производится потоковый ввод и вывод элементов целочисленного массива, содержащего четыре строки и пять столбцов:

```
int b[4][5]; // объявление массива
for (int i = 0; i < 4; ++i) // ввод элементов массива
    for (int j = 0; j < 5; ++j)
    {
        cout << "b[" << i << "][" << j << "] = ";
        cin >> b[i][j];
    }
...
for (int i = 0; i < 4; ++i, cout << endl) // вывод элементов массива
                                                // на экран
    for (int j = 0; j < 5; ++j)
        cout << "b[" << i << "][" << j << "] = " << b[i][j] << "\t";
```

## Замечание

При работе со статическими двумерными массивами можно использовать указатели.

Например:

```
int b[4][5]; // объявление массива
// определяем указатель на тип int и
// устанавливаем его на элемент b[0][0]
int *p = &b[0][0];
for (int i = 0; i < 4; ++i) // ввод элементов массива
    for (int j = 0; j < 5; ++j)
        // введенное значение помещаем по адресу указателя p
        // и сдвигаем указатель на следующий элемент массива
        cin >> *p++;
...
p = &b[0][0]; // повторно устанавливаем указатель на элемент b[0][0]
// вывод элементов массива на экран
for (int i = 0; i < 4; ++i, cout<< endl)
    for (int j = 0; j < 5; ++j)
// выводим на экран значение, хранящееся по адресу
// указателя p, и сдвигаем указатель на следующий элемент массива
    cout << *p++ << "\t";
```

#### Замечание

Напомним, что команда `*p++` с учетом приоритетов операций эквивалентна команде `(*p)++`. При этом вначале будет произведено обращение к значению, которое хранится по адресу, записанному в указатель `p`, а затем этот адрес изменится на столько байт, сколько требуется для размещения в памяти базового типа указателя. Для двумерного массива, в котором элементы расположены последовательно друг за другом (вначале элементы нулевой строки, затем первой строки и т.д.), произойдет переход к адресу следующего элемента массива.

**Динамические двумерные массивы.** Формат объявления динамических массивов:

```
<базовый тип> ** <имя массива>= new <базовый тип> *[<размерность
массива>];
```

#### Замечание

В данном случае квадратные скобки [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.

Например:

```
int **a = new int *[10];
```

#### Замечание

Напомним, динамические массивы отличаются от статических тем, что имя массива является указателем-переменной, а память под элементы массива выделяется программистом в куче с помощью специальной команды `new`.

В данном случае переменная *a* является указателем на массив указателей целого типа. Чтобы из него получить двумерный массив, потребуется выделить память под каждый из 10 указателей. Это можно сделать с помощью цикла:

```
for (int i = 0; i < 10; ++i)
    a[i] = new int [5];
```

В этом случае мы получим двумерный массив целых чисел, который содержит 10 строк и 5 столбцов. В общем случае вместо констант 10 и 5 можно поставить переменные, и тогда получим массив необходимой размерности для данной задачи. Кроме того, для каждого значения *i* мы можем создавать одномерные массивы различной размерности. В результате у нас получится не прямоугольный массив, а свободный (количество элементов в каждой строке различно).

Обратиться к элементам динамического массива можно с помощью индексации, например *a[i][j]*, или с помощью указателей *\*(\*(a+i)+j)*.

Рассмотрим более подробно второе выражение. Указатель *a* хранит адрес указателя на нулевую строку. Операция *a+i* обратится к указателю на *i*-ю строку массива. Выражение *\*(a+i)* позволит обратиться к значению, хранящемуся по адресу *a+i* — фактически к адресу нулевого столбца *i*-й строки двумерного массива. И соответственно, выражение *\*(\*((a+i)+j))* позволит обратиться к значению *j*-го столбца *i*-й строки.

#### Замечание

Как и в случае одномерных массивов, рассмотренные способы обращения к элементам массива эквивалентны друг другу. Однако первый способ более интуитивно понятен, а во втором случае программа будет выполняться быстрее, так как компилятор автоматически переходит от индексного способа обращения к массиву к обращению через указатели.

Ввод и вывод массивов, так же как и в случае статических массивов, производится поэлементно. В следующем фрагменте программы производится потоковый ввод и вывод целочисленного массива, который содержит четыре строки и пять столбцов:

```
int **b = new int *[4]; // объявили двумерный динамический массив
for (int i = 0; i < 2; ++i) b[i] = new int [5];
for (int i = 0; i < 4; ++i) // ввод элементов массива
    for (int j = 0; j < 5; ++j)
    {
        cout << "b[" << i << "][" << j << "] = ";
        cin >> *(*(b+i)+j); // введенное значение записываем в b[i][j]
    }
...
for (int i = 0; i < 4; ++i, cout << endl) // вывод элементов массива
    for (int j = 0; j < 5; ++j)
        // выводим на экран значение b[i][j]
        cout << "b[" << i << "][" << j << "] = " << *(*(b+i)+j) << "\t";
```

Освободить память, выделенную под динамический массив, можно с помощью операции **delete**. Например, освобождение памяти, выделенной под элементы массива *b* из предыдущего примера, производится следующим образом:

```
for (int i = 0; i < 4; ++i)
    // освобождаем память, выделенную под i-ю строку массива
    delete [] b[i];
// освобождаем память, выделенную под массив указателей
delete [] b;
```

### Замечание

Данный фрагмент программы освобождает память, выделенную в куче под двумерный массив, сам указатель *b* при этом сохраняется и может повторно использоваться для определения другого массива.

**Передача двумерных массивов в качестве параметров.** При использовании в качестве параметра двумерного массива в функцию передается указатель на элемент, стоящий в нулевой строке и нулевом столбце. Этим указателем является имя массива. Так же, как и в случае с одномерными массивами, операцию разадресации внутри функции при обращении к элементам массива выполнять не надо, так как запись вида *a[i][j]* автоматически заменяется компилятором на выражение *\*(\*(a+i)+j)*.

При передаче двумерного массива в качестве параметра функции информация о количестве строк и столбцов теряется, поэтому размерности массива следует передавать в качестве отдельных параметров. Рассмотрим несколько примеров.

### Пример 1

Передача статического массива:

```
#include <iostream>
using namespace std;
/* В функцию в качестве параметров передаются статический
массив mas и его размерности: n - количество строк,
m - количество столбцов*/
void print (int mas[3][2], int n, int m)
{
    for (int i = 0; i < n; ++i, cout << endl)
        for (int j = 0; j < m; ++j)
            cout << mas[i][j] << "\t";
}
int main()
{
    int a[3][2] = {{1, -2}, {-3, 4}, {-5, 6} };
    print(a, 3, 2);
    cout << endl;
    print (a, 2, 2)
    return 0;
}
```

Результат работы программы:

```
1 -2  
-3 4  
-5 6
```

```
1 -2  
-3 4
```

При втором вызове функции *print* в качестве параметра *n* мы передали значение, меньшее реального количества строк в массиве. Поэтому мы смогли распечатать не весь массив, а только его часть. Подумайте, что произойдет, если в качестве *n* или *m* передать значение, большее реальной размерности массива.

---

## Пример 2

Передача динамического массива:

```
#include <iostream>  
using namespace std;  
/* В функцию в качестве параметров передаются динамический массив  
mas и его размерности: n - количество строк, m - количество столб-  
цов*/  
void print (int **mas, int n, int m)  
{  
    for (int i = 0; i < n; ++i, cout << endl)  
        for (int j = 0; j < m; ++j)  
            cout << mas[i][j] << "\t";  
}  
int main()  
{  
    int n = 3, m = 4;  
    int **a = new int *[n];  
    for (int i = 0; i < n; ++i)  
        a[i] = new int [m];  
    for (int i = 0; i < n; ++i)  
        for (int j = 0; j < m; ++j)  
            a[i][j] = i+j;  
    print(a, 3, 4);  
    // освобождение памяти  
    for (int i = 0; i < n; ++i)  
        delete [] a[i];  
    delete [] a;  
    return 0;  
}
```

Результат работы программы:

```
0 1 2 3  
1 2 3 4  
2 3 4 5
```

Поскольку массив всегда передается в функцию по адресу, то все изменения, внесенные в формальный параметр массива, отразятся на фактическом параметре массиве. Если необходимо запретить изменение массива в функции, то его нужно передавать как параметр-константу. Например:

```
int sum (const int **mas, const int n)
{ ... }
```

Напомним, что двумерный массив — это одномерный массив указателей, каждый из которых в свою очередь ссылается на одномерный массив. Поэтому двумерный массив можно передавать во вспомогательную функцию для обработки построчно:

```
#include <iostream>
using namespace std;
/* В функцию в качестве параметров передаются одномерный массив
mas и его размерность n */
void print (int *mas, int n)
{
    for (int i = 0; i < n; ++i)
        cout << mas[i] << "\t";
        cout << endl;
}
int main()
{
    int n = 3, m = 4;
    int **a = new int *[n];
    for (int i = 0; i < n; ++i)
        a[i] = new int [m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            a[i][j] = i + j;
    for (int i = 0; i < n; ++i) // обработка двумерного массива
        // построчно:
        // В качестве аргумента функции print
        // передаем указатель на i-ю строку
        print(a[i], m);
    for (int i = 0; i < n; ++i)
        delete [] a[i];
    delete [] a;
    return 0;
}
```

#### Замечание

---

Результат работы программы будет соответствовать предыдущему примеру.

---

**Массив как возвращаемое значение функции.** Двумерный массив может возвращаться в качестве значения функции только как указатель:

```

#include <iostream>
using namespace std;
int **creat(int n, int m)
{
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int [m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            mas[i][j] = i + j;
    return mas;
}

int main()
{
    int n = 5, m = 5;
    int **a = creat(n,m);
    print(a,n,m);
    for (int i = 0; i < n; ++i)
        delete [] a[i];
    delete [] a;
    return 0;
}

```

Результат работы программы:

```

0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

```

## 6.6. Примеры использования двумерных массивов

Двумерные массивы находят свое применение тогда, когда исходные данные представлены в виде таблицы, или когда для хранения данных удобно использовать табличное представление. Рассмотрим примеры использования двумерных массивов. При решении задач мы будем использовать динамические массивы, но обращаться к элементам через индексы. Однако вы можете обращаться к элементам массива и через указатели. Для этого нужно будет заменить выражение  $a[i][j]$  на  $*(*(a + i) + j)$ .

1. В двумерном массиве, элементами которого являются целые числа, подсчитать среднее арифметическое четных элементов массива.

```

#include <iostream>
using namespace std;
// Функция создает и заполняет двумерный массив
int **creat(int &n, int &m)
{
    cout << "n = ";

```

```

    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int[m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
        {
            cout << "mas[" << i << "][" << j << "] = ";
            cin >> mas[i][j];
        }
    return mas;
}
int main()
{
    int n, m, k = 0;
    float s = 0;
    int **a = creat(n, m);
    for (int i = 0; i < n; ++i) // обработка элементов массива
        for (int j = 0; j < m; ++j) {
            if (!(a[i][j] % 2)) // если элемент массива четный,
            {
                // то добавляем его к сумме и увеличиваем количество
                // четных элементов на 1
                s += a[i][j];
                k++;
            }
        }
    if (k)
        cout << s / k;
    else
        cout << " Четных элементов в массиве нет ";
    for (int i = 0; i < n; i++)
        delete[] a[i];
    delete[] a;
    return 0;
}

```

Результат работы программы:

n	m	Массив Anxm	Ответ
2	3	2 1 3 1 3 6 1 3	4.00
3	2	5 1 7 9	Четных элементов в массиве нет

2. Дан двумерный массив, элементами которого являются целые числа. Найти значение максимального элемента массива.

```

#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{

```

```

cout << "n = ";
cin >> n;
cout << "m = ";
cin >> m;
int **mas = new int *[n];
for (int i = 0; i < n; ++i)
    mas[i] = new int [m];
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
    {
        cout << "mas[" << i << "][" << j << "] = ";
        cin >> mas[i][j];
    }
return mas;
}
int main()
{
    int n,m;
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **a = creat(n,m);
    // первоначально в качестве максимального элемента берем a[0][0]
    int max = a[0][0];
    // просматриваем все элементы массива
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            // если очередной элемент больше значения максимального,
            if (a[i][j] > max)
                // то в качестве максимального запоминаем этот элемент
                max = a[i][j];
    cout << "max = " << max;
    for (int i = 0; i < n; i++)
        delete [] a[i];
    delete [] a;
    return 0;
}

```

Результат работы программы:

n	m	Массив Anxm	Ответ
2	3	2 1 3 1 3 6	6

**3.** Данна квадратная матрица, элементами которой являются вещественные числа. Подсчитать сумму элементов главной диагонали.

*Указания по решению задачи.* Для элементов, стоящих на главной диагонали, характерно то, что номер строки совпадает с номером столбца. Этот факт будем учитывать при решении задачи.

```

#include <iostream>
using namespace std;
float **creat(int &n)
{

```

```

cout << "n = ";
cin >> n;
float **mas = new int *[n];
for (int i = 0; i < n; ++i)
    mas[i] = new int[n];
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
    {
        cout << "mas[" << i << "][" << j << "] = ";
        cin >> mas[i][j];
    }
return mas;
}
int main()
{
    int n;
    float **a = creat(n);
    float s = 0;
    // просматриваем все строки массива, добавляем к сумме
    // значение элемента, стоящего на главной диагонали
    for (int i = 0; i < n; i++)
        s += a[i][i];
    cout << " Сумма элементов главной диагонали = " << s;
    for (int i = 0; i < n; i++)
        delete[] a[i];
    delete[] a;
    return 0;
}

```

Результат работы программы:

n	Массив Anxn			Ответ
	2.4	-1.9	3.1	
3	1.1	3.6	-1.2	Сумма элементов главной диагонали = 4.300
	-2.1	4.5	-1.7	

4. Даны прямоугольная матрица, элементами которой являются вещественные числа. Поменять местами ее строки следующим образом: первую строку с последней, вторую с предпоследней и т.д.

*Указания по решению задачи.* Если в массиве  $n$  строк, то нулевую строку нужно поменять с  $n - 1$ , первую строку — с  $n - 2$ ,  $i$ -ю строку — с  $n - i - 1$ .

```

#include <iostream>
using namespace std;
float **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    float **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int [m];
    for (int i = 0; i < n; ++i)

```

```

    for (int j = 0; j < m; ++j)
    {
        cout << "mas[" << i << "][" << j << "] = ";
        cin >> mas[i][j];
    }
    return mas;
}
int main()
{
    int n, m;
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    float **a = creat(n,m);
    float *z;
    // меняются местами i-я и (n-i-1)-я строки
    for (int i = 0; i < (n/2); ++i)
    {
        z = a[i];
        a[i] = a[n-i-1];
        a[n-i-1] = z;
    }
    // вывод измененного массива
    for (int i = 0; i < n; ++i, cout << endl)
        for (int j = 0; j < m; ++j)
            cout << a[i][j] << "\t";
    for (int i = 0; i < n; ++i)
        delete [] a[i];
    delete [] a;
    return 0;
}

```

Результат работы программы:

n	m	Массив Anxm				Ответ			
		2.4	-1.9	3.1	0.0	-2.1	4.5	-1.7	4.9
3	4	1.1	3.6	-1.2	3.7	1.1	3.6	-1.2	3.7
		-2.1	4.5	-1.7	4.9	2.4	-1.9	3.1	0.0

#### Замечание

Обратите внимание на то, что во время перестановки строк цикл перебирает не все строки, а только половину. Если бы перебирались все строки, то никакой перестановки не произошло бы и массив остался неизменным. Подумайте, почему.

5. Даны прямоугольная матрица, элементами которой являются целые числа. Для каждого столбца подсчитать среднее арифметическое его нечетных элементов и записать полученные данные в новый массив.

*Указания по решению задачи.* Массив результатов будет одномерным, а его размерность будет совпадать с количеством столбцов исходной матрицы.

```

#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int [m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
        {
            cout << "mas[" << i << "][" << j << "] = ";
            cin >> mas[i][j];
        }
    return mas;
}
int main()
{
    int n, m;
    int **a = creat(n,m);
    float *b = new float [m]; // создание массива результатов
    for (int j = 0; j < m; ++j) // для каждого j-го столбца
    {
        // обнуляем сумму и количество нечетных элементов столбца
        b[j] = 0;
        int k = 0;
        // перебираем все элементы j-го столбца
        // и находим сумму и количество нечетных элементов
        for (int i = 0; i < n; i++)
            if (a[i][j]%2)
            {
                b[j] += a[i][j];
                k++;
            }
        // если количество нечетных элементов положительное,
        // то вычисляем среднее арифметическое
        if (k)
            b[j] /= k;
    }
    for (int i = 0; i < m; i++) // вывод массива результатов
        cout << b[i] << "\t";
    for (int i = 0; i < n; i++) // удаление массива a
        delete [] a[i];
    delete [] a;
    delete [] b; // удаление массива b
    return 0;
}

```

Результат работы программы:

n	m	Массив Anxm	Ответ
2	3	2 1 3 4 3 6	Вывод результата: 0.00 2.00 3.00

6. Даны матрица  $A$  и вектор  $X$  соответствующих размерностей.  
Нечетные строки матрицы заменить элементами вектора  $X$ .

```
#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int [m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
        {
            cout << "mas[" << i << "][" << j << "] = ";
            cin >> mas[i][j];
        }
    return mas;
}
int main()
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **a = creat(n, m);
    int *b = new int [m];           // создаем массив b
    for (int i = 0; i < m; ++i) // заполняем массив b
        cin >> b[i];
    // каждую нечетную строку массива a заменяем на массив b
    for (int i = 1; i < n; i += 2)
        for (int j = 0; j < m; ++j) a[i][j] = b[j];
    // вывод измененного массива
    for (int i = 0; i < n; ++i, cout << endl)
        for (int j = 0; j < m; ++j)
            cout << a[i][j] << "\t";
    for (int i = 0; i < n; ++i)
        delete [] a[i];
    delete [] a;
    delete [] b;
    return 0;
}
```

Результат работы программы:

n	m	Массив Anxm	Вектор X	Ответ
		-2 1 13		-2 1 13
		1 -3 6		0 4 7
5	3	3 5 1	0 4 7	3 5 1
		3 15 6		0 4 7
		12 4 -3		12 4 -3

7. Даны две матрицы  $A_{m \times n}$  и  $B_{n \times v}$  соответствующих размерностей, элементами которых являются целые числа. Найти произведение этих матриц.

*Указания по решению задачи.* Операция произведения двух матриц определена только для матриц соответствующих размерностей  $A_{m \times n}$  и  $B_{n \times v}$ , т.е. количество столбцов первой матрицы должно совпадать с количеством строк второй. В качестве результата операции получается матрица  $C_{m \times v} = A_{m \times n} \cdot B_{n \times v}$ . Как видим, количество строк матрицы  $C$  совпадает с количеством строк матрицы  $A$ , а количество столбцов матрицы  $C$  равно количеству столбцов матрицы  $B$ . Каждый элемент искомой матрицы  $C$  определяется по формуле  $c[i, j] = \sum_{k=1}^n a[i, k] \cdot b[k, j]$ , где индекс  $i$  принимает значения от 1 до  $m$ , а индекс  $j$  — от 1 до  $v$ .

```
#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int [m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
    {
        cout << "mas[" << i << "][" << j << "] = ";
        cin >> mas[i][j];
    }
    return mas;
}
/*функция в качестве результата возвращает произведение двух матриц
или NULL, если такое произведение вычислить невозможно*/
int **mult(int **a, int na, int ma, int **b, int nb, int mb)
{
    // если размерности матриц не совпадают, то возвращается 0
    if (ma != nb)
        return NULL;
    else
    {
        // иначе создается матрица размерностью na строк и mb столбцов
        int **c = new int *[na];
        for (int i = 0; i < na; ++i)
            c[i] = new int [mb];
        // выполняем подсчет произведения матриц
        for (int i = 0; i < na; ++i)
            for (int j = 0; j < mb; ++j)
        {
            c[i][j] = 0;
            for (int k = 0; k < ma; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

```

        c[i][j] += a[i][k]*b[k][j];
    }
    return c; // в качестве результата возвращаем матрицу с
}
// функция выводит двумерную матрицу на экран в виде таблицы
void print (int **mas, int n, int m)
{
    for (int i = 0; i < n; i++, cout << endl)
        for (int j = 0; j < m; j++)
            cout << mas[i][j] << "\t";
}
// функция освобождает память, выделенную под параметр массив
void deleteMas(int **mas, int n)
{
    for (int i = 0; i < n; i++)
        delete [] mas[i];
    delete [] mas;
}
int main()
{
    int na, ma, nb, mb;
    int **a = creat(na,ma); // создаем и заполняем матрицу a
    int **b = creat(nb, mb); // создаем и заполняем матрицу b
    // вычисляем произведение матриц a и b
    int **c = mult(a, na, ma, b, nb, mb);
    if (c) // если указатель с не пустой,
        print (c, na, mb); // то выводим на экран матрицу с
        else cout << "к данным матрицам операция неприменима";
    // освобождаем память, выделенную под динамические массивы
    deleteMas(a, na);
    deleteMas(b, nb);
    deleteMas(c, na);
    return 0;
}

```

Результат работы программы:

na	ma	Массив Anaxma			nb	mb	Массив	Bnb×mb	Ответ	Cnaxmb	
		-2	1						1	-11 -8	
3	2	1	-3		2	3	0	4	7	-3 13 -11	
		3	5				1	-3	6	5 -3 51	
3	3	-2	1	1			0	4	7	к данным матрицам опе- рация неприменима	
		1	-3	3	2	3			1	-3	
		3	5	4					6		

### Замечание

Обратите внимание на то, что использование вспомогательных функций повышает читаемость программы. Поэтому если вы видите, что один и тот же фрагмент программы будет выполняться несколько раз, то оформите его в виде функции.

## 6.7. Вставка и удаление элементов в массивах

При объявлении массива мы определяем его максимальную размерность, которая в дальнейшем изменена быть не может. Однако с помощью вспомогательной переменной можно контролировать текущее количество элементов, которое не может быть больше максимального.

### Пример

Рассмотрим фрагмент программы:

```
int *a = new int [10];
int n = 5;
for (int i = 0; i < 5; i++)
    a[i] = i*i;
```

В этом случае массив можно представить следующим образом:

$n = 5$	0	1	2	3	4	5	6	7	8	9
$a$	0	1	4	9	16	случайное число				

Так как во время описания был определен массив из 10 элементов, а заполнено только первые пять, то оставшиеся элементы будут заполнены случайными числами.

Что значит *удалить из одномерного массива* элемент с номером 3? Удаление должно привести к физическому «уничтожению» элемента с номером 3 из массива, при этом общее количество элементов должно быть уменьшено. В этом понимании удаления элемента итоговый массив должен выглядеть следующим образом

$0$	$1$	$2$	$4$	$5$	$6$	$7$	$8$	$9$	недопустимое состояние
$a$	0	1	4	16	случайное число				

Такое удаление для массивов *невозможно*, поскольку элементы массива располагаются в памяти последовательно друг за другом, что позволяет организовать индексный способ обращения к массиву.

Однако «удаление» можно смоделировать сдвигом элементов влево и уменьшением значения переменной, которая отвечает за текущее количество элементов в массиве, на единицу:

$n = 4$	0	1	2	3	4	5	6	7	8	9
$a$	0	1	4	16	случайное число					

В общем случае, если мы хотим удалить элемент массива с номером  $k$  (всего в массиве  $n$  элементов, а последний элемент имеет индекс  $n - 1$ ), то нам необходимо произвести сдвиг элементов, начиная с  $(k + 1)$ -го

на одну позицию влево, т.е. на  $k$ -е место поставить  $(k + 1)$ -й элемент, на место  $k + 1$  —  $(k + 2)$ -й элемент, ..., на место  $n - 2$  —  $(n - 1)$ -й элемент. После чего значение  $n$  уменьшить на 1. В этом случае размерность массива не изменится, изменится лишь текущее количество элементов, и у нас создастся ощущение, что элемент с номером  $k$  удален. Рассмотрим данный алгоритм на примере.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "n = ";
    cin >> n; // ввели размерность массива
    int *a = new int [n]; // создали и заполнили массив
    for (int i = 0; i < n; i++)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }
    int k;
    cout << "k = ";
    cin >> k; // ввели номер удаляемого элемента
    // если номер отрицательный или больше номера последнего
    // элемента, то выводим сообщение об ошибке
    if (k < 0 || k > n-1)
        cout << "error";
    // иначе выполняем сдвиг элементов массива
    else
    {
        for (int i = k; i < n-1; i++)
            a[i] = a[i + 1];
        n--; // уменьшаем текущую размерность массива
        for (int i = 0; i < n; i++)
            // выводим измененный массив на экран
            cout << a[i] << "\t";
    }
    delete [] a;
    return 0;
}
```

Рассмотрим теперь операцию удаления в двумерном массиве. Размерность двумерного массива также зафиксирована на этапе объявления массива. Однако при необходимости можно «смоделировать» удаление целой строки в массиве, выполняя сдвиг всех строк, начиная с  $k$ -й на единицу вверх. В этом случае размерность массива не изменится, а текущее количество строк будет уменьшено на единицу. В качестве примера удалим из двумерного массива строку с номером  $k$ .

```
#include <iostream>
using namespace std;
// функция создает и заполняет двумерный массив
int **creat(int &n, int &m)
```

```

{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int [m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
        {
            cout << "mas[" << i << "][" << j << "] = ";
            cin >> mas[i][j];
        }
    return mas;
}
// функция выводит двумерную матрицу на экран в виде таблицы
void print (int **mas, int n, int m)
{
    for (int i = 0; i < n; i++, cout << endl)
        for (int j = 0; j < m; j++)
            cout << mas[i][j] << "\t";
}
// функция освобождает память, выделенную под массив
void deleteMas(int **mas, int n)
{
    for (int i = 0; i < n; i++)
        delete [] mas[i];
    delete [] mas;
}
int main()
{
    int n, m, k;
    int **a = creat(n,m);
    print(a, n, m); // выводим первоначальный массив
    cout << "k = ";
    cin >> k; // вводим номер строки для удаления
/* Если номер строки отрицательный или больше номера последнего
элемента, то выводим сообщение об ошибке */
    if (k < 0 || k > n-1)
        cout << "error";
    else
    {
        // освобождаем память, выделенную для k-й строки массива
        delete [] a[k];
        for (int i = k; i < n-1; ++i)
            // переадресуем указатели, начиная с k-й строки
            a[i] = a[i + 1];
        a[n - 1] = NULL; // указатель на n-1 строку обнуляем
        --n; // уменьшаем текущее количество строк
        print(a, n, m); // выводим измененный массив
    }
    deleteMas(a, n); // освобождаем память
    return 0;
}

```

Результат работы программы:

n	m	k	Исходный массив A4x4	Измененный массив A3x4
			0 0 0 0	0 0 0 0
4	4	1	1 1 1 1	2 2 2 2
			2 2 2 2	3 3 3 3
			3 3 3 3	

Аналогичным образом можно удалить  $k$ -й столбец в двумерном массиве. Однако, так как у нас нет указателей на столбцы массива, сдвиг столбцов нужно будет проводить поэлементно.

```
#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int[m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
        {
            cout << "mas[" << i << "][" << j << "] = ";
            cin >> mas[i][j];
        }
    return mas;
}
void print(int **mas, int n, int m)
{
    for (int i = 0; i < n; i++, cout << endl)
        for (int j = 0; j < m; j++)
            cout << mas[i][j] << "\t";
}
void deleteMas(int **mas, int n)
{
    for (int i = 0; i < n; i++)
        delete[] mas[i];
    delete[] mas;
}
int main()
{
    int n, m, k;
    int **a = creat(n, m); // создаем и заполняем матрицу a
    print(a, n, m); // выводим первоначальный массив
    cout << "k = ";
    cin >> k; // вводим номер столбца для удаления
    if (k < 0 || k > m - 1)
        cout << "error";
    else
    {
        // выполняем поэлементный сдвиг столбцов
```

```

        for (int j = k; j < m - 1; ++j)
            for (int i = 0; i < n; ++i)
                a[i][j] = a[i][j + 1];
        // уменьшаем текущее количество столбцов в массиве
        --m;
        print(a, n, m); // выводим измененный массив
    }
    deleteMas(a, n);
    return 0;
}

```

Результат работы программы:

n	m	k	Исходный массив A4x4	Измененный массив A4x3
4	4	1	0 1 2 3	0 2 3
			0 1 2 3	0 2 3
			0 1 2 3	0 2 3
			0 1 2 3	0 2 3

Вернемся к массиву из примера 1.

n = 5	0	1	2	3	4	5	6	7	8	9
a	0	1	4	9	16	случайное число	случайное число	случайное число	случайное число	случайное число

Подумаем, что значит добавить элемент в одномерный массив в позицию с номером  $k$ ? В этом случае все элементы, начиная с  $k$ -го, должны быть сдвинуты вправо на одну позицию. Однако сдвиг нужно начинать с конца, т.е. на первом шаге на  $n$ -е место поставить  $(n - 1)$ -й элемент, потом на  $(n - 1)$ -е место поставить  $(n - 2)$ -й элемент, ..., наконец, на  $(k + 1)$ -е место вставить  $k$ -й элемент. Таким образом, копия  $k$ -го элемента будет на  $(k + 1)$ -м месте и на  $k$ -е место можно поставить новый элемент. Затем необходимо увеличить текущее количество элементов на 1.

Выполним в данном массиве сдвиг вправо, начиная с позиции  $k = 3$ . В этом случае массив будет выглядеть следующим образом:

k = 3	0	1	2	3	4	5	6	7	8	9
a	0	1	4	9	9	16	случайное число	случайное число	случайное число	случайное число

Теперь в позицию с номером 3 можно поместить новое значение. Текущее количество элементов в массиве становится равным шести. Подумайте, почему сдвиг нужно выполнять с конца массива, а не с начала, как мы это делали в случае удаления элемента из массива?

Рассмотрим программную реализацию данного алгоритма.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "n = ";

```

```

cin >> n; // ввели текущую размерность массива
int m = 2 * n; // определили максимальную размерность массива
int *a = new int[m]; // создали и заполнили массив
for (int i = 0; i < n; i++)
{
    cout << "a[" << i << "] = ";
    cin >> a[i];
}
// ввели номер элемента, в позицию которого будем
// производить вставку
int k;
cout << "k = ";
cin >> k;
if (k < 0 || k > n - 1 || n + 1 > m)
    cout << "error";
/*если введенный номер отрицательный или больше номера последнего
значимого элемента в массиве, или после вставки размерность массива
станет больше допустимой, то выдаем сообщение об ошибке, иначе */
else
{
    int x;
    cout << "x = ";
    cin >> x; // вводим значение для вставки
    for (int i = n; i > k; i--) // выполняем сдвиг в массиве
        a[i] = a[i - 1];
    a[k] = x; // записываем в позицию k значение x
    n++; // увеличиваем текущую размерность массива
    for (int i = 0; i < n; i++) // выводим измененный массив
        cout << a[i] << "\t";
}
return 0;
}

```

Теперь рассмотрим добавление столбца в двумерный массив. Для этого все столбцы после столбца с номером  $k$  передвигаем на 1 столбец вправо. Затем увеличиваем количество столбцов на 1. После этого копия столбца с номером  $k$  будет находиться в столбце с номером  $k + 1$ . И, следовательно,  $k$ -й столбец можно заполнить новыми значениями. Рассмотрим программную реализацию алгоритма.

```

#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    int **mas = new int *[n];
    for (int i = 0; i < n; ++i)
        // определили максимальное количество столбцов
        mas[i] = new int[2 * m];

```

```

for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
    {
        cout << "mas[" << i << "][" << j << "] = ";
        cin >> mas[i][j];
    }
    return mas;
}
void print(int **mas, int n, int m)
{
    for (int i = 0; i < n; i++, cout << endl)
        for (int j = 0; j < m; j++)
            cout << mas[i][j] << "\t";
}
void deleteMas(int **mas, int n)
{
    for (int i = 0; i < n; i++)
        delete[] mas[i];
    delete[] mas;
}
int main()
{
    int n, m, k;
    int **a = creat(n, m); // создаем и заполняем матрицу a
    // m - текущее количество столбцов, m2 - максимально возможное
    int m2 = 2 * m;
    print(a, n, m); // выводим первоначальный массив
    cout << "k = ";
    cin >> k; // вводим номер столбца для добавления
    if (k < 0 || k > m - 1 || m + 1 > m2)
        cout << "error";
    /*если введенный номер столбца отрицательный или больше номера
    последнего значимого столбца в массиве, или после вставки столбца
    размерность массива станет больше допустимой, то выдаем сообщение
    об ошибке, иначе */
    else
    {
        // выполняем поэлементный сдвиг столбцов
        for (int j = m; j > k; --j)
            for (int i = 0; i < n; ++i)
                a[i][j] = a[i][j - 1];
        // увеличиваем текущее количество столбцов в массиве
        ++m;
        // вводим новые данные в k-й столбец
        for (int i = 0; i < n; ++i)
        {
            cout << "a[" << i << "][" << k << "] = ";
            cin >> a[i][k];
        }
        print(a, n, m); // выводим измененный массив
    }
    deleteMas(a, n);
    return 0;
}

```

Результат работы программы:

n	m	k	Исходный массив				Содержимое		Измененный массив			
			A4×4				нового столбца		A4×5			
4	4	1	0	1	2	3	9		0	9	1	2
			0	1	2	3	9		0	9	1	2
			0	1	2	3	9		0	9	1	2
			0	1	2	3	9		0	9	1	2

Аналогичным образом можно провести вставку новой строки в двумерный массив. Однако если вспомнить, что двумерный массив — это одномерный массив указателей на одномерные массивы, например, целых чисел, то поэлементный сдвиг строк можно заменить на сдвиг указателей.

```
#include <iostream>
using namespace std;
int **creat(int &n, int &m)
{
    cout << "n = ";
    cin >> n;
    cout << "m = ";
    cin >> m;
    // определяем максимальное количество строк
    int **mas = new int *[2 * n];
    for (int i = 0; i < n; ++i)
        mas[i] = new int[m];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
        {
            cout << "mas[" << i << "][" << j << "] = ";
            cin >> mas[i][j];
        }
    return mas;
}
void print(int **mas, int n, int m)
{
    for (int i = 0; i < n; i++, cout << endl)
        for (int j = 0; j < m; j++)
            cout << mas[i][j] << "\t";
}
void deleteMas(int **mas, int n)
{
    for (int i = 0; i < n; i++)
        delete[] mas[i];
    delete[] mas;
}
int main()
{
    int n, m, k;
    int **a = creat(n, m); // создаем и заполняем матрицу a
    // n - текущее количество строк, n2 - максимально возможное
    int n2 = 2 * n;
    print(a, n, m); // выводим первоначальный массив
    cout << "k = ";
```

```

    cin >> k; // вводим номер строки для добавления
    if (k < 0 || k > n - 1 || n + 1 > n2)
        cout << "error";
    else
    {
        for (int i = n; i > k; --i) // выполняем сдвиг строк
            a[i] = a[i - 1];
        ++n; // увеличиваем текущее количество строк в массиве
        a[k] =
            // выделяем память под новую строку массива и заполняем ее
            new int[m];
        for (int j = 0; j < m; ++j)
        {
            cout << "a[" << k << "][" << j << "] = ";
            cin >> a[k][j];
        }
        print(a, n, m); // выводим измененный массив
    }
    deleteMas(a, n);
    return 0;
}

```

Результат работы программы:

n	m	k	Исходный массив A4×4	Содержание новой строки	Измененный массив A5×4
			0 0 0 0		0 0 0 0
			1 1 1 1		9 9 9 9
4	4	1	2 2 2 2	9 9 9 9	1 1 1 1
			3 3 3 3		2 2 2 2
					3 3 3 3

## Упражнения

I. Данна последовательность целых чисел.

### Замечание

Задачи из данного пункта решить двумя способами, используя одномерный массив, а затем двумерный.

1. Заменить все положительные элементы противоположными им числами.
2. Заменить все элементы, меньшие заданного числа, этим числом.
3. Заменить все элементы, попадающие в интервал  $[a, b]$ , нулем.
4. Заменить все отрицательные элементы, не кратные трем, противоположными им числами.
5. Все элементы, меньшие заданного числа, увеличить в два раза.
6. Подсчитать среднее арифметическое элементов.
7. Подсчитать среднее арифметическое отрицательных элементов.
8. Подсчитать количество нечетных элементов.
9. Подсчитать сумму элементов, попадающих в заданный интервал.
10. Подсчитать сумму элементов, кратных девяти.

11. Подсчитать количество элементов, не попадающих в заданный интервал.
12. Подсчитать сумму квадратов четных элементов.
13. Вывести на экран номера всех элементов, больших заданного числа.
14. Вывести на экран номера всех нечетных элементов.
15. Вывести на экран номера всех элементов, которые не делятся на семь.
16. Вывести на экран номера всех элементов, не попадающих в заданный интервал.
17. Определить, является ли произведение элементов трехзначным числом.
18. Определить, является ли сумма элементов двухзначным числом.
19. Вывести на экран элементы с четными индексами (для двумерного массива — сумма индексов должны быть четной).
20. Вывести на экран положительные элементы с нечетными индексами (для двумерного массива — первый индекс должен быть нечетным).

II. Данна последовательность из  $n$  действительных чисел.

### Замечание

Задачи из данного пункта решить, используя одномерный массив.

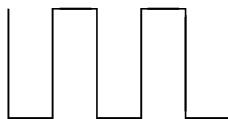
1. Подсчитать количество максимальных элементов.
2. Вывести на экран номера всех минимальных элементов.
3. Заменить все максимальные элементы нулями.
4. Заменить все минимальные элементы на значения с противоположным знаком.
5. Поменять местами максимальный элемент и первый.
6. Вывести на экран номера всех элементов, не совпадающих с максимальным.
7. Найти номер первого минимального элемента.
8. Найти номер последнего максимального элемента.
9. Подсчитать сумму элементов, расположенных между максимальным и минимальным элементами (минимальный и максимальный элементы в массиве единственные). Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
10. Найти номер первого максимального элемента.
11. Найти номер последнего минимального элемента.
12. Подсчитать сумму элементов, расположенных между первым максимальным и последним минимальным элементами. Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
13. Поменять местами первый минимальный и последний максимальный элементы.
14. Найти максимум из отрицательных элементов.
15. Найти минимум из положительных элементов.
16. Найти максимум из модулей элементов.
17. Найти количество пар соседних элементов, разность между которыми равна заданному числу.
18. Подсчитать количество элементов, значения которых больше значения предыдущего элемента.
19. Найти количество пар соседних элементов, в которых предыдущий элемент кратен последующему.
20. Найти количество пар соседних элементов, в которых предыдущий элемент меньше последующего.

III. Дан массив размером  $n \times n$  (если не оговорено иначе), элементы которого — целые числа.

1. Подсчитать среднее арифметическое нечетных элементов, расположенных выше главной диагонали.
2. Подсчитать среднее арифметическое четных элементов, расположенных ниже главной диагонали.
3. Подсчитать сумму элементов, расположенных на побочной диагонали.
4. Подсчитать среднее арифметическое ненулевых элементов, расположенных над побочной диагональю.
5. Подсчитать среднее арифметическое элементов, расположенных под побочной диагональю.
6. Поменять местами столбцы по правилу: первый с последним, второй с предпоследним и т.д.
7. Поменять местами две средние строки, если количество строк четное, и первую со средней строкой, если количество строк нечетное.
8. Поменять местами два средних столбца, если количество столбцов четное, и первый со средним столбцом, если количество столбцов нечетное.
9. Если количество строк в массиве четное, то поменять строки местами по правилу: первую строку со второй, третью — с четвертой и т.д. Если количество строк в массиве нечетное, то оставить массив без изменений.
10. Если количество столбцов в массиве четное, то поменять столбцы местами по правилу: первый столбец со вторым, третий — с четвертым и т.д. Если количество столбцов в массиве нечетное, то оставить массив без изменений.
11. Вычислить  $A^n$ , где  $n$  — натуральное число.
12. Подсчитать норму матрицы по формуле  $\|A\| = \sum_i \max_j a_{i,j}$ .
13. Подсчитать норму матрицы по формуле  $\|A\| = \sum_j \max_i a_{i,j}$ .
14. Вывести элементы матрицы в следующем порядке:

15. Выяснить, является ли матрица симметричной относительно главной диагонали.

16. Заполнить матрицу числами от 1 до  $n$  (где  $n = m \times k$ ,  $m$  — количество строк,  $k$  — количество столбцов прямоугольной матрицы) следующим образом:



17. Определить, есть ли в данном массиве строка, состоящая только из положительных элементов.

18. Определить, есть ли в данном массиве столбец, состоящий только из отрицательных элементов.

19. В каждой строке найти максимум и заменить его противоположным элементом.

20. В каждом столбце найти минимум и заменить его нулем.

**IV.** Дан массив размером  $n \times n$ , элементы которого — целые числа.

1. Найти максимальный элемент в каждой строке и записать данные в новый массив.

2. Найти минимальный элемент в каждом столбце и записать данные в новый массив.

3. Четные столбцы таблицы заменить на вектор  $X$ .

4. Нечетные строки таблицы заменить на вектор  $X$ .

5. Вычислить  $A \cdot X$ , где  $A$  — двумерная матрица,  $X$  — вектор.

6. Для каждой строки подсчитать количество положительных элементов и записать данные в новый массив.

7. Для каждого столбца подсчитать сумму отрицательных элементов и записать данные в новый массив.

8. Для каждого столбца подсчитать сумму четных положительных элементов и записать данные в новый массив.

9. Для каждой строки подсчитать количество элементов, больших заданного числа, и записать данные в новый массив.

10. Для каждого столбца найти первый положительный элемент и записать данные в новый массив.

11. Для каждой строки найти последний четный элемент и записать данные в новый массив.

12. Для каждого столбца найти номер последнего нечетного элемента и записать данные в новый массив.

13. Для каждой строки найти номер первого отрицательного элемента и записать данные в новый массив.

14. Для каждой строки найти сумму элементов с номерами от  $k1$  до  $k2$  и записать данные в новый массив.

15. Для каждого столбца найти произведение элементов с номерами от  $k1$  до  $k2$  и записать данные в новый массив.

16. Для каждой строки подсчитать сумму элементов, не попадающих в заданный интервал, и записать данные в новый массив.

17. Подсчитать сумму элементов каждой строки и записать данные в новый массив. Найти максимальный элемент нового массива.

18. Подсчитать произведение элементов каждого столбца и записать данные в новый массив. Найти минимальный элемент нового массива.

19. Для каждой строки найти номер первой пары неравных элементов. Данные записать в новый массив.

20. Для каждого столбца найти номер первой пары одинаковых элементов. Данные записать в новый массив.

**V.** В одномерном массиве, элементы которого — целые числа, произвести следующие действия.

1. Удалить из массива все четные числа.

2. Удалить из массива все максимальные элементы.

3. Удалить из массива все числа, значения которых попадают в данный интервал.

4. Удалить из массива все элементы, последняя цифра которых равна данной.

5. Удалить из массива элементы с номера  $k1$  по номер  $k2$ .

6. Вставить новый элемент перед первым отрицательным элементом.

7. Вставить новый элемент после последнего положительного элемента.

8. Вставить новый элемент перед всеми четными элементами.

9. Вставить новый элемент после всех элементов, которые заканчиваются на данную цифру.

10. Вставить новый элемент после всех элементов, кратных своему номеру.

11. Удалить из массива все элементы, в записи которых все цифры различны.
12. Удалить из массива повторяющиеся элементы, оставив только их первые вхождения.
13. Вставить новый элемент после всех максимальных.
14. Вставить новый элемент перед всеми элементами, в записи которых есть данная цифра.
15. Вставить новый элемент между всеми парами элементов, имеющими разные знаки.

**VI.** В двумерном массиве, элементы которого — целые числа, произвести следующие действия.

1. Вставить новую строку после строки, в которой находится первый встреченный минимальный элемент.
2. Вставить новый столбец после столбца, в котором нет ни одного отрицательного элемента.
3. Вставить новую строку после всех строк, в которых нет ни одного четного элемента.
4. Вставить новый столбец перед всеми столбцами, в которых встречается заданное число.
5. Вставить строку из нулей после всех строк, в которых нет ни одного нуля.
6. Удалить все строки, в которых нет ни одного четного элемента.
7. Удалить все столбцы, в которых первый элемент больше последнего.
8. Удалить все строки, в которых сумма элементов не превышает заданного числа.
9. Удалить все столбцы, в которых четное количество нечетных элементов.
10. Удалить все строки, в которых каждый элемент попадает в заданный интервал.
11. Удалить все столбцы, в которых все элементы положительны.
12. Удалить все строки, в которых среднее арифметическое элементов является двузначным числом.
13. Удалить строку и столбец, на пересечении которых стоит минимальный элемент (минимальный элемент встречается в массиве только один раз).
14. Удалить из массива  $k$ -ю строку и  $j$ -й столбец, если их значения совпадают.
15. Уплотнить массив, удалив из него все нулевые строки и столбцы.

# Глава 7

## СТРОКИ

Строка — это последовательность символов определенной длины. Существуют два способа (два типа данных), которые используются для представления строк. Первый способ основывается на том, что строка представляет собой массив базового типа `char`, конец которого отмечается нулевым символом '`\0`'. Это старый способ представления строк, унаследованный языком C++ от языка С. Строки данного типа, называемые *строками С*, все еще широко используются. Например, строковые константы в кавычках, такие как "`Hello`", реализуются в C++ как строки С. Кроме того, данный тип используется во многих стандартных библиотеках.

Второй способ основывается на том, что строка представляет собой объект класса `string`. Это современный способ представления строк, который входит в стандарт ANSI/ISO языка C++. Класс `string` позволяет обрабатывать строки посредством стандартных операторов C++, использовать их в составе стандартных выражений. Кроме того, работать с классом `string` гораздо безопаснее, чем с массивом символов, так как в классе `string` контролируются границы массива.

В данном пособии рассмотрим оба способа представления строк.

### Замечание

---

Для работы со строками в стиле С нужно подключить заголовочный файл `cstring`, а для работы с классом `string` нужно подключить заголовочный файл `string`.

---

### 7.1. Работа со строками в виде массивов символов

**Описание и инициализация.** Рассмотрим представление строк в виде массивов типа `char`. Например, строку "Привет" можно представить как массив из семи индексированных переменных — шести букв слова «Привет» и одного дополнительного нулевого символа '`\0`', служащего маркером конца строки. Символ '`\0`' называется *нуль-символом* или *нулевым символом*. Хотя нуль-символ и записывается в виде двух символов, на самом деле он считается одним символом, так же как и символ переноса строки '`\n`'. Использование нуль-символа позволяет обрабатывать массивы посимвольно, корректно определяя конец строки.

Формат объявления строки в виде массива символов:

```
char <имя строки> [n+1];
```

где  $n$  — максимальное количество символов в строке. Единица прибавляется для того, чтобы строка могла вместить не только требуемое количество символов в строке, но и нуль-символ. При этом строка может быть заполнена не до конца, т.е. символов в ней может быть меньше, чем указано в ее описании. Например, объявление

```
char s[20];
```

говорит о том, что в строковую переменную  $s$  можно записать не более 19 символов.

Строковую переменную можно инициализировать при объявлении, при этом не обязательно указывать ее размер. В этом случае размер устанавливается автоматически и будет на единицу больше количества символов, заключенных в кавычки. Например, следующая инициализация:

```
char a[10] = "Привет";
char b[] = "Привет";
```

говорит о том, что в строковой переменной  $a$  заполнено только шесть символов из девяти, а размерность переменной  $b$  равна семи (шесть символов строки "Привет" плюс нуль-символ)

Если же указанный в скобках размер массива символов окажется меньше, чем количество символов в кавычках, то программа выдаст сообщение об ошибке. Например, ошибочным будет описание:

```
char a[5] = "Привет";
```

**Ввод-вывод.** Вывод строк осуществляется обычным образом, т.е с использованием объекта  $cout$  и операции  $<<$ . Например, вывести на экран строковую переменную  $a$ , описанную и инициализированную выше, можно следующим образом:

```
cout << a;
```

Ввод строк можно осуществлять, используя объект  $cin$  и операцию  $>>$ . При этом символы строки будут считываться до тех пор, пока не встретится символ пробела, табуляции или перевода строки, после чего ввод прекратится.

Чтобы ввести строку целиком (до символа перевода строки), необходимо использовать функцию  $getline$  объекта  $cin$ . Функция

```
cin.getline(s, n)
```

осуществляет ввод строки  $s$ , а параметр  $n$  задает максимальное количество символов в этой строке. Если вы введете больше символов, чем задано параметром  $n$ , то лишние символы будут проигнорированы.

Например, ввести строку *s*, объявленную ранее, можно следующим образом:

```
cin.getline(s, 20);
```

### Замечания

1. При указании количества символов в функции *getline* не забывайте про нуль-символ.
2. Если необходимо считать несколько строк, то можно использовать функцию *getline* с тремя аргументами, где третьим аргументом будет символ, на котором заканчивается считывание.

**Основные операции.** Обратиться к символу строки можно так же, как и к элементу обычного массива, указав имя строковой переменной *i*, в квадратных скобках, индекс требуемого символа. При этом нумерация символов в строке, так же как и в обычном массиве, начинается с 0. Рассмотрим на примере, как можно вывести на экран содержимое данной строки, обращаясь к ней посимвольно:

```
char str[20];
cin.getline(str, 20);
int i = 0;
while (str[i] != '\0')
    cout << str[i++] << '\t';
```

При работе с массивом символов надо следить, чтобы символ '\0' не был заменен каким-нибудь другим значением, так как в отсутствие этого символа массив перестанет вести себя как строковая переменная. Хотя программа сможет продолжить работу, но результаты этой работы будут непредсказуемыми, так как произойдет выход за пределы массива и будет обрабатываться какой-то фрагмент памяти, не имеющий отношения к вашей строке.

Строки в стиле С отличаются от переменных других типов данных, и многие операции языка C++ к ним неприменимы. Так, например, нельзя использовать строковую переменную в стиле С в операторе присваивания (хотя при объявлении строковой переменной можно пользоваться знаком равенства для ее инициализации, но больше нигде в программе использовать операцию присваивания не разрешается).

Чтобы присвоить значение строковой переменной в стиле С, можно воспользоваться стандартной функцией *strcpy*:

```
strcpy(s, "Привет"); // присваивает переменной s
                      // значение строки "Привет"
```

Данная версия функции *strcpy* не проверяет, превышает ли размер строки размер строковой переменной. Более безопасной версией присваивания является функция *strncpy*, которая, к сожалению, существует

не во всех реализациях C++. Например, обращение к функции *strncpy* следующим образом:

```
strncpy(s1, s2, 10);
```

копирует 10 символов из строковой переменной *s2* (независимо от ее длины) в строковую переменную *s1*. Например, если в строке *s2* будет содержаться менее 10 символов, то в строку *s1* скопируются только те символы, что имеются.

Проверку эквивалентности двух строковых переменных нельзя выполнять обычным способом, т.е. с помощью оператора `==`. Если применить его для сравнения двух строк, результат окажется неверным, и при этом даже не будет выведено сообщение об ошибке. Сравнение двух строк в стиле C на эквивалентность выполняется с помощью стандартной функции *strcmp*. Функция *strcmp(s1, s2)* возвращает отрицательное число, положительное число или 0 в зависимости от того, окажется первая из переданных ей строк меньше, больше или равной второй с точки зрения их лексикографического порядка.

#### Замечание

Лексикографический (словарный) порядок — это порядок сортировки на основе набора символов ASCII. Если строки состоят лишь из букв, причем либо только строчных, либо только прописных, лексикографический порядок полностью совпадает с обычным алфавитным порядком сортировки.

Если использовать возвращаемый ею результат в качестве логического выражения в операторе *if* или в цикле, то ненулевое значение будет преобразовано в *true*, а 0 — в *false*.

Компиляторы C++, соответствующие стандарту ANSI/ISO, поддерживают более безопасную версию функции *strcmp* с третьим аргументом, в котором задается максимальное количество сравниваемых символов.

Функции *strcpy* и *strcmp* располагаются в библиотеке с заголовочным файлом "cstring".

В табл. 7.1 приведены описания и примеры использования наиболее важных функций для работы со строками в стиле C, которые также расположены в библиотеке с заголовочным файлом "cstring".

**Преобразование строк в числа.** Стока "1234" и число 1234 — это не одно и то же. Первое выражение представляет собой последовательность символов, а второе — целое число. Иногда при работе с числами бывает удобнее считывать их как строки символов, редактировать в строковом виде, а только потом преобразовывать в числа, чтобы выполнить над ними арифметические операции. Кроме того, иногда требуется выделить из строки все числа, которые там имеются, и произвести над ними какие-то арифметические действия.

Для преобразования в число строки в стиле C, состоящей из цифр, используются функции *atoi*, *atol* и *atof*. Для использования данных функций необходимо подключить заголовочный файл "cstdlib".

Таблица 7.1

**Основные функции для работы со строками**

Функция	Описание	Пример	Дополнительная информация
<code>strcpy(s1, s2)</code>	Копирует значение строки <code>s2</code> в строку <code>s1</code>	<pre>char s1[10]; char s2[] = "привет"; strcpy(s1, s2); После этого s1 = "привет"</pre>	Не проверяет, поместится ли значение <code>s2</code> в <code>s1</code>
<code>strncpy(s1, s2, n)</code>	Копирует <code>n</code> символов строки <code>s2</code> в строковую переменную <code>s1</code>	<pre>char s1[10]; char s2[] = "привет"; strncpy(s1, s2, 3); После этого s1 = "при"</pre>	Если значение <code>n</code> выбрано правильно, эта функция надежнее функции <code>strcpy</code> . Реализована не во всех версиях C++
<code>strlen(s)</code>	Возвращает целое число, равное длине строки <code>s</code> (нуль-символ не учитывается)	<pre>char s[] = "привет"; int k; k = strlen(s); После этого k = 6</pre>	
<code>strcat(s1, s2)</code>	Выполняет конкатенацию (слияние) строк, добавляя содержимое строки <code>s2</code> в конец строки <code>s1</code>	<pre>char s1[20] = "Вася, "; char s2[20] = "привет"; strcat(s1, s2); После этого s1 = "Вася, привет", s2 осталась без изменения</pre>	Не проверяет, поместится ли объединенная строка в <code>s1</code>
<code>strncat(s1, s2, n)</code>	Добавляет <code>n</code> символов из <code>s2</code> в конец <code>s1</code>	<pre>char s1[20] = "Вася, "; char s2[20] = "привет"; strncat(s1, s2, 3); После этого s1 = "Вася, при", s2 осталась без изменения</pre>	Если значение <code>n</code> выбрано правильно, то эта функция надежнее <code>strcat</code> . Реализована не во всех версиях C++
<code>strcmp(s1, s2)</code>	Возвращает отрицательное число, положительное число или 0 в зависимости	<pre>char s1[] = "asd"; char s2[] = "afd";</pre>	

Окончание табл. 7.1

Функция	Описание	Пример	Дополнительная информация
<code>strcmp(s1, s2);</code>	от того, окажется <i>s1</i> меньше, больше или равной <i>s2</i> с точки зрения их лексикографического порядка	<pre>char s3[] = "asd"; int k = strcmp(s1, s2); int k1 = strcmp(s1, s3); После этого k = 1 ('s' &lt; 'f'), k1 = 0</pre>	
<code>strncmp(s1, s2, n)</code>	Возвращает отрицательное число, положительное число или 0 в зависимости от того, окажутся ли первые <i>n</i> символов <i>s1</i> меньше, больше или равными первым <i>n</i> символам <i>s2</i> с точки зрения их лексикографического порядка	<pre>char s1[] = "asd"; char s2[] = "afd"; char s3[] = "asd"; int k = strncmp(s1, s2, 1); int k1 = strcmp(s1, s3); После этого k = 0 ("a" = "a"), k1 = 0</pre>	Если значение <i>n</i> выбрано правильно, то эта функция надежнее <code>strcmp</code> . Реализована не во всех версиях C++
<code>strchr(s, ch)</code>	Функция возвращает указатель на первое вхождение символа <i>ch</i> в строке <i>s</i> , если его нет, то возвращает NULL.	<pre>char s[] = "hello"; char ch = 'l'; char *a = strchr(s, ch); *a = 'L'; После этого s = "heLlo"</pre>	

Функция *atoi* принимает один аргумент, строку в стиле С, и возвращает значение типа **int**, соответствующее представленному этой строкой числу. Так,

```
atoi ("1234")
```

возвращает число 1234.

Функция *atol* принимает один аргумент, строку в стиле С, и возвращает значение типа **long**, соответствующее представленному этой строкой числу. Так,

```
atol ("1234567899")
```

возвращает число 1 234 567 899.

Функция *atof* принимает один аргумент, строку в стиле С, и возвращает значение типа **float**, соответствующее представленному этой строкой числу. Так,

```
atof ("12.34")
```

возвращает число 12.34.

При использовании этих функций нужно помнить о следующих моментах:

1) если строка в стиле С, переданная в качестве аргумента любой из функций преобразования, не содержит числа, то функция возвращает 0;

2) если первые символы строки представляют собой число указанного типа, то это число и будет возвращаться в качестве результата, а остальные символы будут проигнорированы.

Например:

```
k = atol("d33.4")      // возвратит 0
k = atol("aa")         // возвратит 0
k = atol("33.4")       // возвратит 33
k = atol("123a45667") // возвратит 123
```

## 7.2. Класс *string*

Класс *string* определен в библиотеке с именем "string" и отнесен к пространству имен *std*. Класс *string* позволяет работать со значениями и выражениями типа *string* почти так же, как со значениями простого типа данных. Так, для присваивания значения переменной типа *string* можно пользоваться оператором *=*, а для конкатенации двух строк — оператором *+*. Предположим, что *s1*, *s2* и *s3* являются объектами типа *string*, и переменные *s1* и *s2* содержат строковые значения. Тогда переменной *s3* можно присвоить строку, состоящую из *s1*, в конец которой добавлена строка *s2*, при помощи оператора присваивания следующим образом:

```
s3 = s1 + s2
```

При этом нет риска, что *s3* окажется слишком «маленькой» для нового значения. Если сумма значений длин строк *s1* и *s2* превысит вместимость переменной *s3*, для нее будет автоматически выделена дополнительная память.

Следует отметить, что заключенные в двойные кавычки строковые литералы являются строками С и не относятся к типу *string*, но их можно использовать для инициализации переменных типа *string*.

**Объявление и инициализация.** Переменную типа *string* можно объявить следующим образом:

```
string <имя переменной>;
```

Например:

```
string s;
```

Поскольку у класса *string* имеется используемый по умолчанию конструктор, который инициализирует объект типа *string* пустой строкой, то данная переменная сразу будет иметь значение «пустая строка».

Кроме того, у класса *string* есть второй конструктор с одним аргументом:

```
string <имя переменной>(<значение>);
```

Например:

```
string s("привет");
```

При таком вызове конструктора строка инициализируется значением, указанным в скобках. При обращении к данному конструктору происходит приведение типов. Заключенная в кавычки строка "привет" является строкой С, а не значением типа *string*. Переменной *s* присваивается объект типа *string*, содержащий те же символы и в том же порядке, что и строка "привет", но его значение не завершается нуль-символом '\0'.

Существует альтернативный синтаксис объявления переменной типа *string* и вызова конструктора этого типа:

```
string s = "привет"; // эквивалентно string s("привет");
```

**Ввод-вывод.** Как и в случае строк в стиле С, объекты класса *string* можно выводить с помощью операции <<, а вводить с помощью операции >>. Например:

```
cin >> s; // ввод строки
cout << s; // вывод строки
```

Ввод значения в строковую переменную закончится тогда, когда встретится символ пробела, табуляции или перевода строки. Если нужно, чтобы программа прочитала всю введенную пользователем строку в переменную типа *string*, можно воспользоваться функ-

цией `getline`. Синтаксис ее использования со строковыми объектами несколько отличается от синтаксиса, описанного для строк в стиле С. В частности, для ввода с клавиатуры вызывается функция `getline` (не являющаяся функцией-членом объекта `cin`), и объект `cin` передается ей в качестве аргумента:

```
getline(cin, s);
```

Данная версия функции прекращает чтение, встретив символ конца строки '`\n`'. Версия функции `getline` с тремя аргументами позволяет задать другой символ, отмечающий конец входных данных. Например, запись:

```
getline(cin, s, '#');
```

указывает, что чтение данных будет прекращено тогда, когда во входном потоке встретится знак `#`.

**Операции над строками.** Класс `string` поддерживает доступ к символам как к элементам массива, т.е. если имеется строка `s`, то, записав `s[i]`, можно обратиться к  $i$ -му символу данной строки.

#### Замечание

---

Нумерация символов в строке, так же как и в массиве, начинается с 0.

---

Допустимость данного индекса никак не контролируется, поэтому если индекс окажется большим, чем количество элементов строки, последствия будут непредсказуемыми (программа может не сообщить об ошибке, но выполнять ошибочные действия, т.е. обратиться в какой-то фрагмент памяти, не имеющий отношения к вашей строке).

Чтобы избежать этой проблемы, можно использовать функцию-член `at`. Функция-член `at` выполняет ту же задачу, что и квадратные скобки, но отличается от них:

- во-первых, синтаксисом — вместо вызова `s[i]` используется вызов `s.at(i)`;
- во-вторых, она проверяет, допустим ли индекс, переданный ей в качестве аргумента. И если значение  $i$  при вызове `s.at(i)` оказывается недопустимым, функция выводит сообщение с указанием ошибки, содержащейся в программе.

Многие операции с объектами этого класса производить удобнее, чем со строками в стиле С. В частности, оператор `==`, выполняемый над объектами класса `string`, возвращает значение `true`, если две строки содержат одинаковые символы в одинаковом порядке, и значение `false` в противном случае. Операторы сравнения `<`, `>`, `<=`, `>=` сравнивают строки с точки зрения их лексикографического порядка.

Кроме того, для работы со строками класса `string` можно использовать операторы присваивания `=` и `+=`:

```
s1 = s2; // в s1 записывается значение s2  
s1+= s2; // в конец s1 добавляется s2
```

Слияние строк можно производить с помощью операции +.

**Длина строки.** Чтобы узнать длину строки, т.е. количество символов в ней, можно воспользоваться функцией-членом *size* или ее синонимом — функцией-членом *length*. Например:

```
string s;  
string::size_type len = s.size();
```

или

```
string::size_type len = s.length();
```

### Замечание

В классе *string* определен вспомогательный тип *size\_type* как синоним типа *unsigned long* или *unsigned int*, т.е. достаточно большой, чтобы содержать размер любой строки. Чтобы воспользоваться этим типом, можно применить оператор области видимости :: .

Чтобы выяснить, является ли строка пустой, можно проверить, равна ли ее длина 0, или использовать функцию-член *empty*, которая возвращает значение *true*, если строка пуста, и *false* в противном случае.

**Выделение подстроки.** Для выделения подстроки из исходной строки используется функция *substr*.

Функция *s.substr(i, n)* возвращает подстроку строки *s*, которая начинается с позиции *i* и содержит *n* символов.

Функция *s.substr(i)* возвращает подстроку строки *s*, которая начинается с позиции *i* и заканчивается последним символом строки *s*.

**Сравнение строк.** Как уже говорилось, в классе *string* определены все операторы, которые позволяют выяснить равенство (==) или неравенство (!=) двух строк, а также сравнивать их (<, <=, >, >=). Кроме операторов сравнения, класс *string* предоставляет набор перегруженных версий функции *compare*, которые осуществляют лексикографическое сравнение.

Пусть определены следующие типы:

```
string s, s1;  
char sc[20];
```

В табл. 7.2 приведены различные варианты сравнения строк с использованием функции *compare*.

**Поиск в строке.** Класс *string* предоставляет шесть вариантов функции поиска. Все они возвращают либо значение типа *string::size\_type*, которое является индексом найденного элемента, либо специальное значение *string::npos*, если ничего не найдено. Каждый из вариантов функции поиска существует в нескольких версиях, которые отличаются

набором аргументов. В табл. 7.3 приведены различные варианты функции поиска.

*Таблица 7.2*

**Варианты использования функции compare**

Обращение к функции	Описание
<code>s.compare(s1)</code>	Возвратит 0, если $s$ и $s1$ равны, отрицательное число, если $s < s1$ , и положительное, если $s > s1$
<code>s.compare(sc)</code>	Аналогичным образом сравнивает строку $s$ со строкой $sc$ в стиле С
<code>s.compare(i, n, s1)</code>	Сравнивает $n$ символов строки $s$ , начиная с позиции $i$ , со строкой $s1$
<code>s.compare(i, n, s1, i1, n1)</code>	Сравнивает $n$ символов строки $s$ , начиная с позиции $i$ , с $n1$ символами строки $s1$ , начиная с позиции $i1$
<code>s.compare(i, n, sc, n1)</code>	Сравнивает $n$ символов строки $s$ , начиная с позиции $i$ , с $n1$ символами строки $sc$ в стиле С

*Таблица 7.3*

**Варианты использования функции find**

Обращение к функции	Описание
<code>s.find(args)</code>	Ищет первое вхождение аргумента в строку $s$
<code>s.rfind(args)</code>	Ищет первое справа вхождение аргумента в строку $s$
<code>s.find_first_of(args)</code>	Ищет первое вхождение любого символа аргумента в строку $s$
<code>s.find_last_of(args)</code>	Ищет последнее вхождение любого символа аргумента в строку $s$
<code>s.find_first_not_of(args)</code>	Ищет первое вхождение символа, который отсутствует в аргументе, в строке $s$
<code>s.find_last_not_of(args)</code>	Ищет последнее вхождение символа, который отсутствует в аргументе, в строке $s$

В качестве аргументов функций поиска могут использоваться:

- $c$  — в исходной строке выполняется поиск символа  $c$ ;
- $c, pos$  — в исходной строке выполняется поиск символа  $c$ , начиная с позиции  $pos$ ;
- $sub$  — в исходной строке выполняется поиск подстроки  $sub$ ;
- $sub, pos$  — в исходной строке выполняется поиск подстроки  $sub$ , начиная с позиции  $pos$ ;

Например:

```
string s = "в данной строке ищем подстроку";
string sub = "строк";
char c = 'o';
```

```

size_type i1 = s.find(sub);           // i1 = 9
size_type i2 = s.find(c);            // i2 = 6
size_type i3 = s.rfind(sub);         // i3 = 24
size_type i4 = s.find(c, 16);        // i4 = 22
size_type i5 = s.find_first_of(sub); // i5 = 6

```

**Изменение строки.** Для замены подстроки в строке существуют различные версии функции *replace*, в которых указывается часть строки, которую нужно заменить, и заменяющий ее символ или набор символов. Заменяемая часть строки задается индексом первого символа и количеством символов. Заменяющий набор символов задается строкой, массивом символов или отдельным символом. Все функции меняют строку и возвращают указатель на нее.

Например:

```
s.replace(i, n, s1);
```

заменяет *n* символов строки *s*, начиная с позиции *i*, на подстроку *s1*.

Добавить подстроку можно, воспользовавшись операцией +. Например:

```
s += s1;
```

добавляет в конец строки *s* значение строки *s1*.

Кроме этого существуют несколько версий функции *append*, добавляющих подстроку, массив символов или отдельный символ в конец строки. Все функции меняют строку и возвращают указатель на нее. Например:

```

s.append(s1);      // добавляет к строке s строку s1;
s.append(n, c);   // добавляет символ с n раз;
s.append(s1, i, n); // добавляет к строке s n символов
                   // строки s1, начиная с i-го

```

Вставить подстроку можно с помощью функции *insert*. Существует несколько версий этой функции, которые предназначены для вставки строки, массива символов или отдельного символа в указанное место данной строки. Место вставки обычно указывается первым аргументом — это индекс элемента строки, перед которым будет сделана вставка. При вставке строка раздвигается, ее длина увеличивается. Функции возвращают ссылку на измененную строку. Например:

```

s.insert(i,s1);    // вставляет в строку s строку s1,
                  // начиная с позиции i
s.insert(i, n,c); // вставляет в строку s символ с n раз,
                  // начиная с позиции i
s.insert(i,s1,j,k); // вставляет в строку s, начиная с позиции i,
                    // k символов строки s1, начиная с j-го

```

Удалить подстроку можно с помощью функции *erase*. Например:

```
s.erase(i,n); // удаляет из строки s n символов, начиная с позиции i
```

В табл. 7.4 содержится описание рассмотренных ранее функций-членов класса *string*. Напомним, чтобы воспользоваться ими, следует подключить заголовочный файл "string".

Таблица 7.4

**Функции-члены класса *string***

Обращение к функции	Описание
<code>s.length()</code>	Возвращает количество символов в строке <i>s</i>
<code>s.substr(i, n)</code>	Возвращает подстроку строки <i>s</i> , которая начинается с позиции <i>i</i> и содержит <i>n</i> символов
<code>s.empty()</code>	Возвращает <i>true</i> , если <i>s</i> является пустой строкой, и <i>false</i> в противном случае
<code>s.insert(i, s1)</code>	Вставляет в строку <i>s</i> строку <i>s1</i> , начиная с позиции <i>i</i>
<code>s.erase(i, n)</code>	Удаляет из строки <i>s</i> <i>n</i> символов, начиная с позиции <i>i</i>
<code>s.find(s1)</code>	Возвращает индекс первого вхождения строки <i>s1</i> в строку <i>s</i>
<code>s.find(s1, i)</code>	Возвращает индекс первого вхождения строки <i>s1</i> в строку <i>s</i> , причем поиск начинается с позиции <i>i</i>
<code>s.rfind(s1)</code>	Возвращает индекс первого справа вхождения строки <i>s1</i> в строку <i>s</i>
<code>s.rfind(s1, i)</code>	Возвращает индекс первого справа вхождения строки <i>s1</i> в строку <i>s</i> , причем поиск ведется, начиная с позиции <i>i</i>
<code>s.replace(i, n, s1)</code>	Заменяет <i>n</i> символов строки <i>s</i> , начиная с позиции <i>i</i> , на подстроку <i>s1</i>
<code>s.compare(s1)</code>	Возвращает 0, если <i>s</i> = <i>s1</i> , отрицательное число, если <i>s</i> < <i>s1</i> , и положительное, если <i>s</i> > <i>s1</i>

### 7.3. Взаимное преобразование объектов типа *string* и строк в стиле C

Как мы уже видели, C++ выполняет автоматическое преобразование типов, позволяющее присваивать строки в стиле C переменным типа *string*. Например, приведенный ниже код является корректным:

```
char s1[] = "привет";
string s2;
s2 = s1;
```

А вот использование следующего оператора недопустимо:

```
s1 = s2; // ошибка: несовместимость типов
```

Недопустим и следующий оператор:

```
strcpy(s1, s2); // ошибка: несовместимость типов
```

Это происходит потому, что функция `strcpy` не может принимать в качестве второго аргумента объект типа `string` и C++ не выполняет автоматического преобразования объектов `string` в строки C, что является реальной проблемой.

Для получения строки в стиле C, соответствующей объекту `string`, нужно выполнить явное преобразование типов. Это можно сделать с использованием функции-члена `c_str`. Вот как правильно произвести копирование объекта `string` в строку C:

```
strcpy(s1, s2.c_str());
```

## 7.4. Работа с отдельными символами

В табл. 7.5 приведен список функций, применимых к символьным значениям. Все они проверяют переданный им символ и возвращают значение `true`, если проверка пройдена, и `false` в противном случае.

Таблица 7.5

Функции, применимые к символьным значениям

Обращение к функции	Описание
<code>isalpha(c)</code>	Возвращает <code>true</code> , если <code>c</code> — буква
<code>islower(c)</code>	Возвращает <code>true</code> , если <code>c</code> — символ нижнего регистра
<code>isupper(c)</code>	Возвращает <code>true</code> , если <code>c</code> — символ верхнего регистра
<code>isalnum(c)</code>	Возвращает <code>true</code> , если <code>c</code> — буква или цифра
<code>isdigit(c)</code>	Возвращает <code>true</code> , если <code>c</code> — цифра
<code>isxdigit(c)</code>	Возвращает <code>true</code> , если <code>c</code> — шестнадцатеричная цифра
<code>isprint(c)</code>	Возвращает <code>true</code> , если <code>c</code> — печатаемый символ, включая пробел
<code>ispunct(c)</code>	Возвращает <code>true</code> , если <code>c</code> — знак пунктуации
<code>isspace(c)</code>	Возвращает <code>true</code> , если <code>c</code> — пробельный символ (пробел, табуляция, возврат каретки, перевод страницы)
<code>isgraph(c)</code>	Возвращает <code>true</code> , если <code>c</code> — не пробел, а печатаемый символ
<code>iscntrl(c)</code>	Возвращает <code>true</code> , если <code>c</code> — управляемый символ
<code>tolower(c)</code>	Если <code>c</code> — прописная буква, возвращает ее эквивалент в нижнем регистре, в противном случае оставляет без изменений
<code>toupper(c)</code>	Если <code>c</code> — строчная буква, возвращает ее эквивалент в верхнем регистре, в противном случае оставляет без изменений

## 7.5. Смешанный строко-числовой ввод данных

Рассмотрим следующий пример программы:

```
#include "iostream"
#include "string"
using namespace std;
int main()
{
    int age;
    string name, address;
    cout << "Name: ";
    getline(cin, name);
    cout << "Age: ";
    cin >> age;
    cout << "Address: ";
    getline(cin, address);
    cout << "\n" << name << "\t" << age << "\t" << address << endl;
}
```

В данном примере используется смешанный ввод данных — для ввода числовых информации применяется операция `>>`, а для ввода строковых данных — функция `getline`. Результат работы этой программы будет выглядеть примерно так:

```
Name: Petrov
Age: 23
Address:
Petrov 23
Press any key to continue
```

Вы никогда не получите возможность ввести адрес. Проблема состоит в том, что `>>` читает до пробельного символа (пробела, табуляции, перехода на новую строку), оставляя сам пробельный символ в потоке. В данном случае будет прочитан возраст, а символ новой строки '`\n`', сгенерированный нажатием клавиши *Enter*, останется во входном потоке. Функция `getline` воспримет символ '`\n`', оставшийся во входном потоке, как пустую строку, и запишет ее в переменную `address`. Исправить эту ошибку можно, только читая символ новой строки перед чтением данных в строковую переменную `address`. Это можно сделать, используя функцию `get` без параметров для чтения одного символа из входного потока:

```
cin.get();
```

Преобразуем нашу программу так, чтобы все данные вводились корректно:

```
#include "iostream"
#include "string"
using namespace std;
int main()
```

```

{
    int age;
    string name, address;
    cout << "Name: ";
    getline(cin, name);
    cout << "Age: ";
    cin >> age;
    cout << "Address: ";
    cin.get(); // считывание символа \n из входного потока
    getline(cin, address);
    cout << "\n" << name << "\t" << age << "\t" << address << endl;
}

```

## 7.6. Примеры работы со строками

1. Даны строка и символ. Требуется выяснить, сколько раз данный символ встречается в данной строке. Рассмотрим несколько вариантов решения данной задачи:

а) с помощью строки в стиле C, используя простой перебор символов:

```

#include "iostream"
#include "cstring"
using namespace std;
int main()
{
    char str[20];
    char symbol;
    int k = 0;
    cout << "Enter string" << endl;
    cin.getline(str, 20);
    cout << "Enter symbol" << endl;
    cin >> symbol;
    for (unsigned int i = 0; i < strlen(str); i++)
        if (str[i] == symbol)
            k++;
    cout << "k = " << k << endl;
    return 0;
}

```

б) с помощью объекта класса *string*, используя простой перебор символов:

```

#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str;
    char symbol;
    int k = 0;
    cout << "Enter string" << endl;
    getline(cin, str);

```

```

cout << "Enter symbol" << endl;
cin >> symbol;
for (unsigned int i = 0; i < str.length(); i++)
    if (str[i] == symbol)
        k++;
cout << "k = " << k << endl;
return 0;
}

```

в) с помощью объекта класса *string*, используя функцию *find*:

```

#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str;
    char symbol;
    int k = 0;
    // позиция, начиная с которой ищем символ
    string::size_type pos = 0;
    cout << "Enter string" << endl;
    getline(cin, str);
    cout << "Enter symbol" << endl;
    cin >> symbol;
    // номер первого вхождения
    string::size_type n = str.find(symbol, pos);
    // пока номер вхождения не равен pros,
    // т.е. пока данный символ в строке есть
    while (n != string::npos)
    {
        // счетчик увеличиваем на 1 и начинаем искать новое вхождение
        k++;
        // с позиции, следующей за найденной, чтобы каждый раз
        // не находить одно и то же
        pos = n + 1;
        n = str.find(symbol, pos);
    }
    cout << "k = " << k << endl;
    return 0;
}

```

2. В данной строке заменить все вхождения одного символа на другой:

а) с помощью строки в стиле С, используя функцию *strchr*:

```

#include "iostream"
#include "cstring"
using namespace std;
int main()
{
    char str[20];
    char symbol, symbol1;
    int k = 0;
    cout << "Enter string" << endl;
    cin.getline(str, 20);

```

```

cout << "Enter symbol" << endl;
cin >> symbol;
cout << "Enter symbol" << endl;
cin >> symbol1;
// устанавливаем указатель на первое вхождение
// символа symbol в строке str
char *s = strchr(str, symbol);
while (s != NULL) // пока указатель не пустой
{
    *s = symbol1; // заменяем данное значение на введенный символ
    // перемещаем указатель на заданный символ в строке str
    s = strchr(str, symbol);
}
cout << "str = " << str << endl;
return 0;
}

```

б) с помощью объекта класса *string*, используя функцию *find*:

```

#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str;
    char symbol1, symbol2;
    cout << "Enter string" << endl;
    getline(cin, str);
    cout << "Enter symbol 1" << endl;
    cin >> symbol1;
    cout << "Enter symbol 2" << endl;
    cin >> symbol2;
    string::size_type k = str.find(symbol1);
    while (k != string::npos) {
        str[k] = symbol2;
        k = str.find(symbol1);
    }
    cout << "string = " << str << endl;
    return 0;
}

```

3. В данной строке заменить каждую точку многоточием:

#### **Замечание**

---

Используем следующий алгоритм: находим первое вхождение точки в строку, вставляем после нее еще две точки. Затем снова ищем первое вхождение точки в строку, но уже с позиции *pos*, т.е. с позиции, следующей за последней вставленной точкой.

---

```

#include "iostream"
#include "string"
using namespace std;
int main()

```

```

{
    string str;
    string::size_type k = 0, pos = 0;
    cout << "Enter string" << endl;
    getline(cin, str);
    k = str.find(".", pos);
    while (k != string::npos)
    {
        str.insert(k + 1, "..");
        pos = k + 3;
        k = str.find(".", pos);
    }
    cout << "string = " << str << endl;
    return 0;
}

```

4. Удалить из заданной строки все цифры:

а) используем тот факт, что все символы упорядочены, следовательно, если данный символ является цифрой, он должен находиться между символами '0' и '9'.

```

#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str;
    unsigned int k = 0, pos = 0;
    cout << "Enter string" << endl;
    getline(cin, str);
    while (k < str.length()) {
        if (str[k] >= '0' && str[k] <= '9')
            str.erase(k, 1);
        else
            k++;
    }
    cout << "string = " << str << endl;
    return 0;
}

```

б) используем функцию *isdigit*, которая возвращает значение *true*, если ее аргумент цифра.

```

#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str;
    unsigned int k = 0, pos = 0;
    cout << "Enter string" << endl;
    getline(cin, str);
    while (k < str.length())
    {

```

```

    if (isdigit(str[k]))
        str.erase(k, 1);
    else
        k++;
}
cout << "string = " << str << endl;
return 0;
}

```

**5.** Данна строка, которая представляет собой слова, разделенные пробелами. Вывести на экран количество слов в строке, которые начинаются и заканчиваются одной и той же буквой.

#### Замечание

---

Используем следующий алгоритм: находим первое вхождение пробела в строку и выделяем подстроку от первого символа до первого пробела. Получаем первое слово. Затем пересчитываем номер позиции, с которой начинаем поиск, и повторяем описанные действия.

---

```

#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str, slovo;
    int s = 0;
    string::size_type k = 0, pos = 0;
    cout << "Enter string" << endl;
    getline(cin, str);
    str = str + ' ';
    k = str.find(" ", pos);
    while (k != string::npos)
    {
        slovo = str.substr(pos, k - pos);
        if (slovo[0] == slovo[slovo.length() - 1])
            s++;
        pos = k + 1;
        k = str.find(" ", pos);
    }
    cout << "kolichestvo = " << s << endl;
    return 0;
}

```

**6.** Рассмотрим усложненный вариант предыдущей задачи, когда слова в строке могут разделяться пробелами и знаками препинания. Предложение заканчивается точкой.

#### Замечание

---

Используем вариант функции поиска, который ищет первый из набора символов. Еще одно отличие от предыдущей программы заключается в том, что если после слова стоит знак пунктуации, то в следующей позиции будет

пробел, а не начало нового слова. А пробел тоже входит в набор искомых символов (*razdel*), поэтому в этом случае надо перейти на две позиции вперед (*pos = k + 2*).

---

```
#include "iostream"
#include "string"
using namespace std;
int main()
{
    string str, slovo;
    int s = 0;
    string::size_type k = 0, pos = 0;
    string razdel = ",.:;!? ";
    cout << "Enter string" << endl;
    getline(cin, str);
    k = str.find_first_of(razdel, pos);
    while (k != string::npos)
    {
        slovo = str.substr(pos, k - pos);
        if (ispunct(str[k]))
            pos = k + 2;
        else
            pos = k + 1;
        if (slovo[0] == slovo[slovo.length() - 1])
            s++;
        k = str.find_first_of(razdel, pos);
    }
    cout << "kolichestvo = " << s << endl;
    return 0;
}
```

## Упражнения

**I.** Поставьте знак сравнения (*>*, *<*, *==*) между парами строк и обоснуйте свой ответ:

- 1) "Процессор" \_\_\_\_ "Процесс";
- 2) "Balkon" \_\_\_\_ "balkon";
- 3) "кошка" \_\_\_\_ "кошка";
- 4) "окно" \_\_\_\_ "оКно";
- 5) "муха" \_\_\_\_ "СЛОН";
- 6) "кино" \_\_\_\_ "кино";
- 7) " мышь" \_\_\_\_ "мышь";
- 8) "Иванов" \_\_\_\_ "Петров";
- 9) "Смирнов" \_\_\_\_ "Смирнова".

**II.** Простые действия со строками.

1. В данной строке вставить символ *c1* после каждого вхождения символа *c2*.
2. Вставить после каждого вхождения подстроки *str1* подстроку *str2*.
3. Определить, сколько в строке гласных букв (при условии, что текст записан кириллицей).
4. Выяснить, имеются ли в строке два соседствующих одинаковых символа.
5. Удвоить каждое вхождение заданной буквы в строке.

6. Удалить из строки все символы  $c1$ .
7. Удалить из строки все подстроки  $str1$ .
8. Заменить в строке все вхождения подстроки  $str1$  на подстроку  $str2$ .
9. Определить, сколько различных символов встречается в строке.
10. В строке имеются только две одинаковые буквы. Найти их.
11. Данна строка. Подсчитать количество содержащихся в ней прописных латинских букв.
12. Даны целые положительные числа  $N1$  и  $N2$  и строки  $S1$  и  $S2$ . Получить из этих строк новую строку, содержащую первые  $N1$  символов строки  $S1$  и последние  $N2$  символов строки  $S2$  (в указанном порядке).
13. Даны символ  $C$  и строки  $S$ ,  $S0$ . Перед каждым вхождением символа  $C$  в строку  $S$  вставить строку  $S0$ .
14. Даны символ  $C$  и строки  $S$ ,  $S0$ . После каждого вхождения символа  $C$  в строку  $S$  вставить строку  $S0$ .
15. Даны строки  $S$  и  $S0$ . Найти количество вхождений строки  $S0$  в строку  $S$ .
16. Даны строки  $S$  и  $S0$ . Удалить из строки  $S$  все подстроки, совпадающие с  $S0$ . Если совпадающих подстрок нет, то вывести строку  $S$  без изменений.
17. Данна строка, содержащая по крайней мере один символ пробела. Вызвести подстроку, расположенную между первым и вторым пробелом исходной строки. Если строка содержит только один пробел, то вывести пустую строку.
18. Данна строка, содержащая по крайней мере один символ пробела. Вызвести подстроку, расположенную между первым и последним пробелом исходной строки. Если строка содержит только один пробел, то вывести пустую строку.
19. Данна строка-предложение. Зашифровать ее, поместив вначале все символы, расположенные на четных позициях строки, а затем, в обратном порядке, все символы, расположенные на нечетных позициях (например, строка «Программа» превратится в «ргамамроП»).

**III. Сложные действия со строками.** Дано осмысленное текстовое сообщение (т.е. алфавитно-цифровая информация, разделенная пробелами и знаками препинаний, в конце которого ставится точка).

1. Подсчитать, сколько раз заданное слово встречается в сообщении (при этом заданное слово не может являться частью другого слова).
2. Найти самое длинное слово в сообщении.
3. Найти самое короткое слово в сообщении.
4. Вызвести на экран все слова-палиндромы, содержащиеся в заданном сообщении.
5. Подсчитать количество слов, содержащихся в сообщении.
6. Вызвести на экран все слова сообщения, состоящие из  $n$  букв.
7. Вызвести только те слова сообщения, которые начинаются на заданную букву.
8. Вызвести только те слова сообщения, которые начинаются и оканчиваются на одну и ту же букву.
9. Вызвести только те слова сообщения, которые начинаются и оканчиваются на заданную букву.
10. Вызвести только те слова сообщения, которые встречаются в нем ровно один раз.
11. Вызвести только те слова сообщения, которые встречаются в нем более  $n$  раз.
12. Вызвести все слова сообщения, которые содержат данную букву.
13. После каждого заданного слова в сообщении поставить восклицательный знак.
14. Преобразовать сообщение так, чтобы каждое его слово начиналось с заглавной буквы.

15. Поменять слова в сообщении по принципу: первое со вторым, третье с четвертым и т.д.

16. Поменять слова в сообщении по принципу: первое с последним, второе с предпоследним и т.д.

17. Поменять слова в сообщении по принципу: первое с  $(n/2 + 1)$ -словом, второе — с  $(n/2 + 2)$ -словом,  $i$ -е с  $(n/2 + i)$ -словом и т.д ( $n$  — число слов в предложении).

18. Удалить из сообщения все однобуквенные слова.

19. Удалить из сообщения все слова, начинающиеся с заглавной буквы.

20. Удалить из сообщения все повторяющиеся слова.

#### IV. Преобразование символов в числа.

1. Дан текст, содержащий цифры. Вывести на экран все имеющиеся в нем цифры.

2. Дан текст, содержащий цифры. Вывести на экран только нечетные цифры.

3. Дан текст, содержащий цифры. Вывести на экран количество цифр в нем.

4. Дан текст, содержащий цифры. Вывести на экран их сумму.

5. Дан текст, содержащий цифры. Вывести на экран наибольшую цифру.

6. Дан текст, содержащий цифры. Найти наибольшее количество идущих подряд цифр.

7. Дан текст, содержащий цифры. Заменить все нечетные цифры наименьшей цифрой, содержащейся в данном тексте.

8. Дан текст, имеющий вид:  $d_1 + d_2 + \dots + d_n$ , где  $d_i$  — цифры. Вычислить записанную в тексте сумму.

9. Дан текст, имеющий вид:  $d_1 - d_2 + d_3 - \dots$ , где  $d_i$  — цифры. Вычислить значение данного выражения.

10. Дан текст, имеющий вид:  $d_1 \pm d_2 \pm \dots \pm d_n$ , где  $d_i$  — цифры. Вычислить значение данного выражения.

11. Дан текст, содержащий цифры. Найти наибольшее количество идущих подряд цифр.

12. Дан текст. Определить, является ли он правильной десятичной записью целого числа.

13. Дан текст, представляющий собой десятичную запись целого числа. Вычислить сумму цифр этого числа.

14. Дан текст, содержащий целые числа. Вывести на экран все имеющиеся в нем числа.

15. Дан текст, содержащий целые числа. Вывести на экран только четные числа.

16. Дан текст, содержащий целые числа. Вывести на экран количество чисел в нем.

17. Дан текст, содержащий целые числа. Вывести на экран сумму нечетных чисел.

18. Дан текст, содержащий целые числа. Вывести на экран наименьшее из имеющихся чисел.

19. Дан текст. Определить, является ли он правильной десятичной записью вещественного числа.

20. Дан текст, содержащий вещественные числа. Вывести на экран все вещественные числа, содержащиеся в нем.

## Самостоятельная работа

**Задача 1.** Известны фамилия, имя и отчество пользователя. Найти его код личности. Правило получения кода личности: каждой букве ставится в соответ-

ствие число — порядковый номер буквы в алфавите. Эти числа складываются. Если полученная сумма не является однозначным числом, то цифры числа снова складываются и так до тех пор, пока не будет получено однозначное число. Например:

*Исходные данные:* Александр Сергеевич Пушкин

*Код личности:*  $(1 + 13 + 6 + 12 + 19 + 1 + 15 + 5 + 18) + (19 + 6 + 18 + 4 + 6 + 6 + 3 + 10 + 25) + (17 + 21 + 26 + 12 + 10 + 15) = 288 \Rightarrow 2 + 8 + 8 = 18 \Rightarrow 1 + 8 = 9.$

**Задача 2.** В шифре Цезаря алфавит размещается на круге по часовой стрелке. За последней буквой алфавита идет первая буква алфавита, т.е. после буквы «я» идет буква «а». При шифровании текста буквы заменяются другими буквами, отстоящими по кругу на заданное количество позиций (сдвиг) дальше по часовой стрелке. Например, если сдвиг равен 3, то буква «а» заменяется на букву «г», буква «б» на букву «д», а буква «я» на букву «в».

Зашифровать сообщение, используя шифр Цезаря со сдвигом  $k$ .

Например, дан алфавит  $\{a, b, c\}$ , с помощью которого записано слово *abaccc*. Используя шифр Цезаря со сдвигом 2, мы получим слово *cacbba*.

# Глава 8

## РЕКУРСИВНЫЕ ФУНКЦИИ. ПЕРЕГРУЗКА ФУНКЦИЙ И ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ

Напомним, что в гл. 2 «Функции в C++» было введено понятие функций, а также рассмотрены различные способы описания, определения и вызова функций. В данной главе мы продолжим изучение данной темы.

### 8.1. Рекурсивные функции

*Рекурсивной* называют функцию, если она вызывает сама себя в качестве вспомогательной. В основе рекурсивной функции лежит так называемое рекурсивное определение какого-либо понятия. Классическим примером рекурсивной функции является функция, вычисляющая факториал.

Из курса математики известно, что  $0! = 1! = 1$ ,  $n! = 1 \cdot 2 \cdot 3 \cdots \cdot n$ . С другой стороны,  $n! = (n - 1)! \cdot n$ . Таким образом, известны два значения параметра  $n$ , а именно  $n = 0$  и  $n = 1$ , при которых мы без каких-либо дополнительных вычислений можем определить значение факториала. Во всех остальных случаях, т.е. для  $n > 1$ , значение факториала может быть вычислено через значение факториала для  $n - 1$ . Таким образом, рекурсивная функция будет иметь вид

```
#include "iostream"
using namespace std;
unsigned long F(short n) // рекурсивный метод
{
    if (n == 0 || n == 1)
        return 1; // нерекурсивная ветвь
    // шаг рекурсии - повторный вызов функции с другим параметром
    else
        return n * F(n - 1);
}

int main()
{
    short n;
    cout << "n = ";
    cin >> n;
```

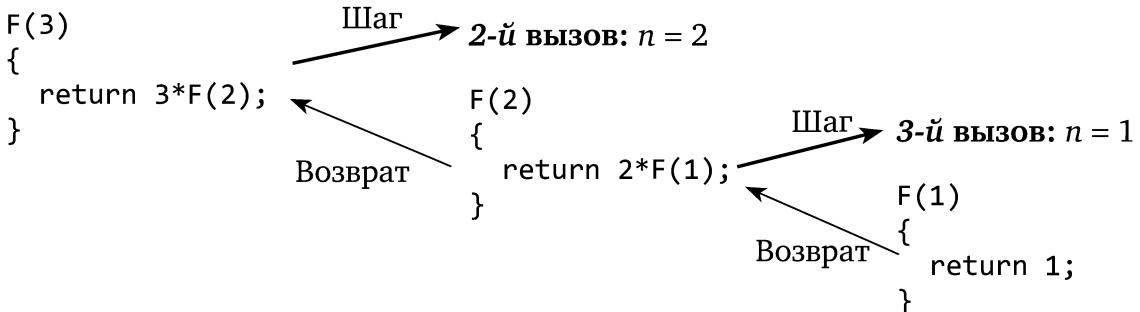
```

unsigned long f = F(n); // вызов рекурсивной функции F
cout << n << "!=" << f << endl;
return 0;
}

```

Рассмотрим работу описанной выше рекурсивной функции для  $n = 3$ .

**1-й вызов:**  $n = 3$



Первый вызов функции  $F$  осуществляется из функции *main*. В нашем случае это происходит во время выполнения оператора присваивания  $f = F(3)$ . Этап входления в рекурсию обозначим жирными стрелками. Он продолжается до тех пор, пока значение переменной  $n$  не становится равным 1. После этого начинается выход из рекурсии (тонкие стрелки). В результате вычислений получается, что  $F(3) = 3 * 2 * 1$ .

Рассмотренный вид рекурсии называют прямым. Метод с прямой рекурсией обычно содержит следующую структуру:

```

if (<условие>)
    <оператор>;
else <вызов данной функции с другими параметрами>;

```

В качестве <условия> записываются некоторые граничные случаи параметров, передаваемых рекурсивной функции, при которых результат ее работы заранее известен. Далее следует простой оператор или блок, а в ветви **else** происходит рекурсивный вызов данной функции с другими параметрами.

Что необходимо знать для реализации рекурсивного процесса? Со входом в рекурсию осуществляется вызов функции, а для выхода необходимо помнить точку возврата, т.е. то место программы, откуда мы пришли и куда нам нужно будет возвратиться после завершения выполнения функции. Место хранения точек возврата называется стеком вызовов, и для него выделяется определенная область оперативной памяти. В стеке запоминаются не только адреса точек возврата, но и копии значений всех параметров, по которым при возврате восстанавливается вызывающая функция. Из-за создания копий параметров при развертывании рекурсии может произойти переполнение стека. Это является основным недостатком рекурсивной функции. Другой недостаток — это время, которое затрачивается на вызовы функции. Вместе с тем рекурсивные функции позволяют перейти к более компактной записи алгоритма.

Любую рекурсивную функцию можно преобразовать в обычную функцию. И наоборот, практически любую функцию можно преобразовать в рекурсивную, если выявить рекуррентное соотношение между вычисляемыми с помощью функции значениями.

Далее каждую задачу будем решать с использованием обычной и рекурсивной функций.

### Пример 1

---

Найти сумму цифр числа  $A$ .

Известно, что любое натуральное число  $A = a_n a_{n-1} \dots a_1 a_0$ , где  $a_n, a_{n-1}, \dots, a_0$  — цифры числа, можно представить следующим образом:

$$\begin{aligned} A &= a_n a_{n-1} \dots a_1 a_0 = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0 = \\ &= ((\dots((a_n \cdot 10 + a_{n-1}) \cdot 10 + a_{n-2}) \cdot 10 \dots) \cdot 10 + a_1) \cdot 10 + a_0. \end{aligned}$$

Например, число 1234 можно представить как

$$1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 = ((1 \cdot 10 + 2) \cdot 10 + 3) \cdot 10 + 4.$$

Из данного представления видно, что получить последнюю цифру можно, если найти остаток от деления числа на 10. В связи с этим для разложения числа на составляющие его цифры можно использовать следующий алгоритм:

- 1) находим остаток при делении числа  $A$  на 10, т.е. получаем крайнюю правую цифру числа;
- 2) находим целую часть числа при делении  $A$  на 10, т.е. отбрасываем от числа  $A$  крайнюю правую цифру;
- 3) если преобразованное  $A > 0$ , то переходим к п. 1. Иначе число равно нулю, и отделять от него больше нечего.

Данный алгоритм будет использоваться при разработке нерекурсивной функции.

С другой стороны, сумму цифр числа 1234 можно представить следующим образом:  $\text{sum}(1234) = \text{sum}(123) + 4 = (\text{sum}(12) + 3) + 4 = (((\text{sum}(1) + 2) + 3) + 4) = (((\text{sum}(0) + 1) + 2) + 3) + 4$ . При этом если  $A = 0$ , то сумма цифр числа также равна нулю, т.е.  $\text{sum}(0) = 0$ . В противном случае сумму цифр числа  $A$  можно представить рекуррентным соотношением  $\text{sum}(A) = \text{sum}(A/10) + A \% 10$ . Полученное рекуррентное соотношение будем использовать при разработке рекурсивной функции.

```
#include "iostream"
using namespace std;
unsigned long F(unsigned long a) // нерекурсивная функция
{
    unsigned long sum = 0;
    while (a > 0) // пока a больше нуля
    {
        sum += a % 10; // добавляем к сумме последнюю цифру числа a
        a /= 10; // отбрасываем от числа a последнюю цифру
    }
    return sum; // возвращаем в качестве результата сумму цифр
               // числа a
}
unsigned long F_Rec(unsigned long a) // рекурсивная функция
{
```

```

if (a == 0)
    return 0; // если a = 0, то возвращаем 0
// иначе обращаемся к рекуррентному соотношению
else
    return F_Rec(a / 10) + a%10;
}
int main()
{
    unsigned long a;
    cout << "a = ";
    cin >> a;
    cout << "F(" << a << ") = " << F(a) << endl;
    cout << "F_Rec(" << a << ") = " << F_Rec(a) << endl;
    return 0;
}

```

---

## Пример 2

Вычислить  $n$ -й член последовательности Фибоначчи.

Первые два члена последовательности Фибоначчи равны 1, остальные получаются по рекуррентной формуле  $a_n = a_{n-1} + a_{n-2}$ .

```

#include "iostream"
using namespace std;
unsigned long F(short n) // нерекурсивная функция
{
    if (n == 1 || n == 2)
        return 1;
    else
    {
        unsigned long a, a1 = 1, a2 = 1;
        for (short i = 3; i <= n; ++i)
        {
            a = a1 + a2;
            a2 = a1;
            a1 = a;
        }
        return a;
    }
}
unsigned long F_Rec(short n) // рекурсивная функция
{
    if (n == 1 || n == 2)
        return 1;
    else
        return F_Rec(n - 1) + F_Rec(n - 2);
}
int main()
{
    short n;
    cout << "n = ";
    cin >> n;
    cout << "F(" << n << ") = " << F(n) << endl;

```

```
    cout << "F_Rec(" << n << ") = " << F_Rec(n) << endl;
    return 0;
}
```

---

Рассмотренные выше рекурсивные функции возвращали некоторое значение, заданное рекуррентным соотношением. Однако не все функции возвращают значение. Кроме того, рассмотренные выше функции представляют собой простой вариант рекурсивной функции. В общем случае рекурсивная функция может включать в себя некоторое множество нерекурсивных операторов и один или несколько операторов рекурсивного вызова. Рассмотрим примеры «сложных» рекурсивных функций, не возвращающих значение.

### Пример 3

---

Для заданного значения  $n$  вывести на экран  $n$  строк, в каждой из которых содержится  $n$  звездочек. Например, для  $n = 5$  на экран нужно вывести следующую таблицу:

```
/*
**
***
****

#include "iostream"
using namespace std;
// функция выводит на экран строку из n звездочек
void Stroka(short n)
{
    for (short i = 1; i <= n; ++i)
        cout << '*';
    cout << endl;
}
void F(short n) // нерекурсивная функция
{
    // выводит n строк по i звездочек в каждой
    for (short i = 1; i <= n; ++i)
        Stroka(i);
}
// рекурсивная функция, где i - номер текущей строки,
// n - номер последней строки, при этом номер строки
// совпадает с количеством звездочек в строке
void F_Rec(short i, short n)
{
    // если номер текущей строки не больше номера
    // последней строки, то
    if (i <= n)
    {
        Stroka(i); // выводим i звездочек в текущей строке и
        // переходим к формированию следующей строки
        F_Rec(i + 1, n);
    }
}
```

```

int main()
{
    short n;
    cout << "n = ";
    cin >> n;
    cout << "_____ F _____ \n";
    F(n); //
    cout << "_____ F_Rec _____ \n";
    // 1 - номер первой строки, n - номер последней строки
    F_Rec(1, n);
    return 0;
}

```

---

#### Пример 4

Для заданного нечетного значения  $n$  (например, для  $n = 7$ ) вывести на экран следующую таблицу:

```

*****
 ****
 ***
 *
 *
 ***
 ****
*****

```

Данную таблицу условно можно разделить на две части. Рассмотрим отдельно верхнюю часть.

Номер строки	Содержимое экрана	$i$ — количество пробелов в строке	Количество звездочек в строке
0	*****	0	7
1	****	1	5
2	***	2	3
3	*	3	1

Таким образом, если нумеровать строки с нуля, то номер строки совпадает с количеством пробелов, которые нужно напечатать в начале этой строки. Так как количество звездочек в каждой следующей строке уменьшается на 2, то количество звездочек в строке с номером  $i$  можно определить по формуле  $n - 2i$ , где  $n$  — это количество звездочек в нулевой строке. Всего нужно напечатать  $n/2 + 1$  строк.

Аналогичную зависимость можно выявить и для нижней части таблицы.

```

#include "iostream"
using namespace std;
// функция выводит на экран n раз символ a
void Stroka(short n, char a)
{
    for (short i = 1; i <= n; ++i)
        cout << a;
}

```

```

void F(short n) // нерекурсивная функция
{
    // выводим верхнюю часть таблицы, в которой в каждой строке
    for (short i = 0; i <= n / 2; ++i)
    {
        Stroka(i, ' ');           // вначале печатаем пробелы
        Stroka(n - 2 * i, '*'); // затем звездочки
        cout << endl; // затем переводим курсор на новую строку
    }
    // аналогично выводим нижнюю часть таблицы
    for (int i = n / 2; i >= 0; --i)
    {
        Stroka(i, ' ');
        Stroka(n - 2 * i, '*');
        cout << endl;
    }
}
// рекурсивная функция, где i определяет номер текущей
// строки, n - количество звездочек в строке
void F_Rec(short i, short n)
{
    if (n > 0)
    {
        // действия до рекурсивного вызова
        // позволяют вывести верхнюю часть таблицы
        Stroka(i, ' ');
        Stroka(n, '*');
        cout << endl;
        // вызываем эту же функцию, увеличивая номер строки
        // и уменьшая количество звездочек в ней
        F_Rec(i + 1, n - 2);
        // действия после рекурсивного вызова
        // позволяют вывести нижнюю часть таблицы
        Stroka(i, ' ');
        Stroka(n, '*');
        cout << endl;
    }
}
int main()
{
    short n;
    cout << "n = ";
    cin >> n;
    cout << " _____ F _____ \n";
    F(n);
    cout << " _____ F_Rec _____ \n";
    F_Rec(0, n);
    return 0;
}

```

---

Все примеры, рассмотренные ранее, относились к прямой рекурсии. Однако существует еще и косвенная рекурсия, в которой функция вызывает себя через другую вспомогательную функцию. Продемонстрируем

косвенную рекурсию на примере программы, которая для заданного значения  $n$  выводит на экран следующее за ним простое число.

Данная программа содержит функцию *Prim*, которая возвращает *true*, если ее параметр является простым числом, *false* — в противном случае. Чтобы установить, является ли число  $j$  простым, нужно проверить делимость числа  $j$  на все простые числа, не превышающие квадратный корень из  $j$ . Перебор таких простых чисел можно организовать следующим образом: рассмотреть первое простое число, 2, а затем, используя функцию *NextPrim*, возвращающее следующее за значением ее параметра простое число, получить все простые числа, не превышающие квадрата числа  $j$ . В свою очередь, функция *NextPrim* обращается к функции *Prim*, чтобы определить, является ли заданное число простым.

Таким образом, функции *Prim* и *NextPrim* перекрестно вызывают друг друга. В этом и проявляется косвенная рекурсия.

```
#include "iostream"
using namespace std;
int NextPrim(int i); // опережающее описание
bool Prim(int j)
{
    int k = 2; // первое простое число
    // значение k «пробегает» последовательность простых чисел,
    // начиная с 2 до корня из j, при этом проверяется, делится ли j
    // на одно из таких простых чисел
    while (k * k <= j && j % k != 0)
        k = NextPrim(k); // вызов функции NextPrim
    return (j % k == 0) ? false : true;
}
int NextPrim(int i)
{
    int p = i + 1;
    while (!Prim(p)) // вызов функции Prim
        ++p;
    return p;
}
int main()
{
    int n;
    cout << "n = ";
    cin >> n;
    cout << "Следующее за " << n << " простое число равно "
        << NextPrim(n) << endl;
    return 0;
}
```

### Замечание

Обратите внимание на то, что при объявлении функции *NextPrim* использовалось «опережающее описание», смысл которого заключается в том, что заголовок функции указывается ранее, чем полностью описывается сама функция. Такое описание позволяет обращаться к функции *NextPrim* из функции *Prim*.

Рекурсия является удобным средством решения многих задач: сортировки числовых массивов, обхода таких структур данных, как деревья и графы.

С другой стороны, применение рекурсивных функций в ряде случаев оказывается нерациональным. Вспомним рекурсивную функцию подсчета  $n$ -го члена последовательности Фибоначчи. Данная функция будет работать весьма неэффективно.  $F_{Rec}(17)$  вычисляется в ней как  $F_{Rec}(16) + F_{Rec}(15)$ . В свою очередь,  $F_{Rec}(16)$  вычисляется в ней как  $F_{Rec}(15) + F_{Rec}(14)$ . Таким образом,  $F_{Rec}(15)$  будет вычисляться два раза,  $F_{Rec}(14)$  — три раза,  $F_{Rec}(13)$  — пять раз и т.д. Всего для вычисления  $F_{Rec}(17)$  потребуется выполнить более тысячи операций сложения. Для сравнения, при вычислении  $F(17)$ , т.е. используя нерекурсивную функцию подсчета  $n$ -го члена последовательности Фибоначчи, потребуется всего лишь 15 операций сложения.

Таким образом, при разработке рекурсивных функций следует задуматься об их эффективности.

## 8.2. Перегрузка функций

Иногда бывает удобно, чтобы функции, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Использование нескольких функций с одним и тем же именем, но различными типами и количеством параметров называется *перегрузкой*. Компилятор определяет, какую именно функцию требуется вызвать, по типу и количеству фактических параметров.

Рассмотрим следующий пример:

```
#include "iostream"
using namespace std;
int max(int a) // первая версия функции max
{
    int b = 0;
    while (a > 0)
    {
        if (a % 10 > b)
            b = a % 10;
        a /= 10;
    }
    return b;
}
int max(int a, int b) // вторая версия функции max
{
    if (a > b)
        return a;
    else
        return b;
}
int max(int a, int b, int c) // третья версия функции max
{
```

```

if (a > b && a > c)
    return a;
else if (b > c)
    return b;
else
    return c;
}
int main()
{
    int a = 1283, b = 45, c = 35740;
    cout << max(a) << endl;
    cout << max(a, b) << endl;
    cout << max(a, b, c) << endl;
}

```

При вызове функции *max* компилятор выбирает вариант, соответствующий типу и количеству передаваемых в функцию параметров. Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если выбор перегруженной функции возможен более чем одним способом, то выбирается «лучший» из вариантов (вариант, содержащий меньшее количество и длину преобразований в соответствии с правилами преобразования типов). Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

#### Замечание

Перегрузка функций является проявлением *полиморфизма*, одного из основных свойств объектно-ориентированного программирования (ООП). Программисту гораздо удобнее помнить одно имя функции и использовать его для работы с различными типами данных, а решение о том, какой вариант функции вызвать, возложить на компилятор. Этот принцип широко используется в стандартных библиотеках C++. Например, функция *pow* заголовочного файла "math" имеет семь перегруженных версий: *pow(double x, double y)*, *pow(double x, int y)*, *pow(float x, float y)* и т.д.

## 8.3. Функции-шаблоны

Многие алгоритмы не зависят от типов данных, с которыми они работают. Например, функция, вычисляющая максимальное значение в одномерном массиве, должна «уметь» обрабатывать как целочисленные, так и вещественные массивы. Для решения этой проблемы в C++ можно применить перегрузку функций. Однако в этом случае нам придется написать девять функций — шесть для целых типов и три для вещественных, что не только увеличит объем программы, но и снизит ее быстродействие, так как компилятор будет тратить время на выбор нужной версии функции. В C++ существует другая возможность решения данной проблемы — через использование функций-шаблонов.

С помощью функций-шаблонов можно реализовать алгоритм, который будет применяться к различным типам данных, а конкретный тип данных будет передаваться в функцию в качестве параметра на этапе компиляции. Компилятор автоматически генерирует код, соответствующий переданному типу.

Формат функции-шаблона:

```
template <typename имя_типа>
заголовок_функции
{
    тело_функции
}
```

где *template* используется для обозначения функции-шаблона; *<typename имя\_типа>* определяет параметр, который будет хранить информацию о том, какой именно тип данных был передан в шаблон; *заголовок\_функции* определяется стандартным образом, т.е указывается тип возвращаемого значения, имя функции и, в круглых скобках, список параметров; при этом тип возвращаемого значения и (или) хотя бы один из параметров функции должен иметь тип, определенный параметром *имя\_типа*.

Рассмотрим использование функций-шаблонов на следующем примере:

```
#include "iostream"
using namespace std;
// функция-шаблон для вывода одномерного массива на экран
template <typename X> void printArray(char *name, X *a, int n)
{
    cout << name;
    for (int i = 0; i < n; ++i)
        cout << a[i] << "\t";
    cout << endl;
}
// функция-шаблон для поиска максимального
// значения в одномерном массиве
template <typename X> X maxArray(X *a, int n)
{
    X max = a[0];
    for (int i = 0; i < n; ++i)
        if (max < a[i])
            max = a[i];
    return max;
}
int main()
{
    int a[] = {1, -2, 4, -5, 3};
    // обращение к функциям-шаблонам с целочисленным массивом
    int maxA = maxArray(a, 5);
    printArray("Array a: ", a, 5);
    cout << "max(a) = " << maxA << endl;
    // обращение к функциям-шаблонам с вещественным массивом
```

```

double b[] = {3.4, -3.1, -6.5, 2};
double maB = maxArray(b, 4);
printArray("Array b: ", b, 4);
cout << "max(b) = " << maB << endl;
}

```

В функции-шаблоне *printArray* параметр *X* используется для хранения информации о типе массива, который передается в данную функцию в качестве второго параметра.

В функции-шаблоне *maxArray* параметр *X* используется для хранения информации о типе массива, который передается в данную функцию в качестве первого параметра, а также для определения типа возвращаемого значения данной функции. При этом параметр *X* используется для описания переменной *max* в теле функции *maxArray*.

В общем случае функция-шаблон может содержать несколько параметров, отвечающих за информацию о типах данных. Например:

```

template <typename One, typename Two>
void F(One a, Two b)
{ ... }

```

Чтобы более полно использовать механизм шаблонов, нужно изучить основы ООП.

## Упражнения

### I. Разработка нерекурсивных функций.

1. Разработать функцию, которая для заданного натурального числа *N* возвращает значение *true*, если число простое, и значение *false*, если число составное. С помощью данной функции:

- 1) вывести на экран все простые числа на отрезке  $[a, b]$ ;
- 2) найти количество всех простых чисел на отрезке  $[a, b]$ ;
- 3) найти сумму всех составных чисел на отрезке  $[a, b]$ ;
- 4) для заданного числа *A* вывести на экран предшествующее по отношению к нему простое число.

2. Разработать функцию, которая для заданного натурального числа *N* возвращает количество его делителей. С помощью данной функции:

- 1) для каждого натурального числа на отрезке  $[a, b]$  вывести на экран количество делителей;
- 2) вывести на экран только те натуральные числа отрезка  $[a, b]$ , у которых количество делителей равно заданному числу;
- 3) вывести на экран только те натуральные числа отрезка  $[a, b]$ , у которых количество делителей максимально;
- 4) для заданного числа *A* вывести на экран следующее по отношению к нему число, имеющее столько же делителей, сколько и число *A*.

3. Разработать функцию, которая для заданного натурального числа *N* возвращает сумму его делителей. С помощью данной функции:

- 1) для каждого натурального числа на отрезке  $[a, b]$  вывести на экран сумму его делителей;
- 2) вывести на экран только те натуральные числа отрезка  $[a, b]$ , у которых сумма делителей равна заданному числу;

3) вывести на экран только те натуральные числа отрезка  $[a, b]$ , у которых сумма делителей максимальна.

4) для заданного числа  $A$  вывести на экран предшествующее по отношению к нему число, сумма делителей которого равна сумме делителей числа  $A$ .

4. Разработать функцию, которая для заданного натурального числа  $N$  возвращает сумму его цифр. С помощью данной функции:

1) для каждого целого числа на отрезке  $[a, b]$  вывести на экран сумму его цифр;

2) вывести на экран только те целые числа отрезка  $[a, b]$ , у которых сумма цифр числа равна заданному значению;

3) вывести на экран только те целые числа отрезка  $[a, b]$ , у которых сумма цифр нечетная;

4) для заданного числа  $A$  вывести на экран предшествующее по отношению к нему число, сумма цифр которого равна сумме цифр числа  $A$ .

5. Разработать функцию, которая для заданных натуральных чисел  $N$  и  $M$  возвращает их наибольший общий делитель. С помощью данной функции:

1) сократить дробь вида  $a/b$ ;

2) найти наименьшее общее кратное для двух натуральных чисел;

3) вычислить значение выражения  $\frac{a}{b} + \frac{d}{c}$ ; результат представить в виде обыкновенной дроби, выполнив сокращение;

4) найти наибольший общий делитель для  $n$  натуральных чисел.

II. Разработать рекурсивную функцию, возвращающую значение:

1) для вычисления  $n$ -го члена следующей последовательности:

$$b_1 = -10, \quad b_2 = 2, \quad b_{n+2} = |b_n| - 6b_{n+1};$$

2) для вычисления  $n$ -го члена следующей последовательности:

$$b_1 = 5, \quad b_{n+1} = \frac{b_n}{n^2 + n + 1};$$

3) для вычисления количества цифр в заданном натуральном числе;

4) для нахождения наибольшего общего делителя методом Евклида:

$$\text{НОД}(a, b) = \begin{cases} a, & \text{если } a = b; \\ \text{НОД}(a - b, b), & \text{если } a > b; \\ \text{НОД}(a, b - a), & \text{если } b > a; \end{cases}$$

5) для вычисления значения функции Аккермана для неотрицательных чисел  $n$  и  $m$ . Функция Аккермана определяется следующим образом:

$$A(n, m) = \begin{cases} m + 1, & \text{если } n = 0; \\ A(n - 1, 1), & \text{если } n \neq 0, m = 0; \\ A(n - 1, A(n, m - 1)), & \text{если } n > 0, m > 0; \end{cases}$$

6) для вычисления числа сочетаний  $C(n, m)$ , где  $0 \leq m \leq n$ , используя следующие свойства:

$$C_n^0 = C_n^n = 1; \quad C_n^m = C_{n-1}^m + C_{n-1}^{m-1} \text{ при } 0 < m < n;$$

7) для вычисления числа  $a$ , для которого выполняется неравенство  $2^{a-1} \leq n \leq 2^a$ , где  $n$  — натуральное число. Для подсчета числа  $a$  использовать формулу

$$a(n) = \begin{cases} 1, & \text{если } n = 1; \\ a(n / 2) + 1, & \text{если } n > 1; \end{cases}$$

8) для вычисления  $x^n$  ( $x$  — вещественное,  $x \neq 0$ , а  $n$  — целое) по формуле

$$x^n = \begin{cases} 1 & \text{при } n = 0, \\ 1/x^{|n|} & \text{при } n < 0, \\ x \cdot x^{n-1} & \text{при } n > 0 \end{cases}$$

вычислить значение  $x^n$  для различных  $x$  и  $n$ ;

9) для вычисления  $\sum_{i=1}^n i$ , где  $n$  — натуральное число. Для заданных натуральных чисел  $m$  и  $k$  вычислить с помощью разработанного метода значение выражения  $\sum_{i=1}^m i + \sum_{i=1}^{2k} i$ ;

10) для вычисления значения функции  $F(N) = \frac{N}{\sqrt{1+\sqrt{2+\sqrt{3+\dots+\sqrt{N}}}}.$

Найти ее значение при заданном натуральном  $N$ ;

11) для вычисления цепной дроби:

$$\cfrac{x}{1+\cfrac{x}{2+\cfrac{x}{3+\dots\cfrac{x}{n+x}}}}.$$

Найти значение данной дроби при заданном натуральном  $n$ ;

12) для вычисления суммы цифр в строке; с помощью данной функции определить, в каком из двух предложений сумма цифр больше;

13) для вычисления количества цифр в строке; с помощью данной функции определить, в каком из двух предложений цифр больше;

14) определяющую, является ли заданная строка палиндромом;

15) определяющую, является ли палиндромом часть строки  $s$ , начиная с  $i$ -го символа и заканчивая  $j$ -м символом;

16) для перевода числа из десятичной системы счисления в двоичную;

17) для перевода числа из двоичной системы счисления в десятичную;

18) для вычисления наибольшего значения в одномерном массиве;

19) для вычисления наименьшего значения в двумерном массиве;

20) для вычисления суммы элементов в одномерном массиве.

**III. Разработать рекурсивную функцию, не возвращающую значений.**

1. Даны первый член и разность арифметической прогрессии. Написать рекурсивную функцию для нахождения  $n$ -го члена и суммы  $n$  первых членов прогрессии.

2. Даны первый член и знаменатель геометрической прогрессии. Написать рекурсивную функцию для нахождения  $n$ -го члена и суммы  $n$  первых членов прогрессии.

3. Разработать рекурсивную функцию, которая по заданному натуральному числу  $N$  выведет на экран в порядке возрастания все натуральные числа, не превышающие  $N$ . Например, для  $N = 8$  на экран выводится 1 2 3 4 5 6 7 8.

4. Разработать рекурсивную функцию, которая по заданному натуральному числу  $N$  выведет на экран все натуральные числа, не превышающие  $N$ , в порядке убывания. Например, для  $N = 8$  на экран выводится 8 7 6 5 4 3 2 1.

5. Разработать рекурсивную функцию для вывода на экран стихотворения:

10 лунатиков жили на луне

10 лунатиков ворочались во сне

Один из лунатиков упал с луны во сне  
 9 лунатиков осталось на луне  
 9 лунатиков жили на луне  
 9 лунатиков ворочались во сне  
 Один из лунатиков упал с луны во сне  
 8 лунатиков осталось на луне  
 .....  
 И больше лунатиков не стало на луне

6. Дано натуральное число  $n$ . Разработать рекурсивную функцию для вывода на экран следующей последовательности чисел:

```

1
2   2
3   3   3
...
n   n   n   ...   n
  
```

7. Дано натуральное число  $n$ . Разработать рекурсивную функцию для вывода на экран следующей последовательности чисел:

```

1
2   1
3   2   1
...
n   n - 1   n - 2   ...   1
  
```

8. Разработать рекурсивную функцию для вывода на экран цифр натурального числа в прямом порядке. Применить эту функцию ко всем числам из интервала от  $A$  до  $B$ .

9. Разработать рекурсивную функцию для вывода на экран всех делителей заданного натурального числа  $n$ .

10. Дано натуральное четное число  $n$ . Разработать рекурсивную функцию для вывода на экран следующей картинки:

*****	$n$ звездочек
*****	$n - 1$ звездочка
*****	$n - 2$ звездочки
...	
*	1 звездочка

11. Дано натуральное четное число  $n$ . Разработать рекурсивную функцию для вывода на экран следующей картинки:

*****	0 пробелов, $n$ звездочек
*****	1 пробел, $n - 1$ звездочка
*****	2 пробела, $n - 2$ звездочки
...	
*	$n - 1$ пробел, 1 звездочка

12. Дано натуральное четное число  $n$ . Разработать рекурсивную функцию для вывода на экран следующей картинки:

*	*	$n$ пробелов между звездочками
**	**	$n - 2$ пробела
***	***	$n - 4$ пробела
...		...
*****	*****	2 пробела
*****		0 пробелов
*****	*****	2 пробела
...		...
***	***	$n - 4$ пробела
**	**	$n - 2$ пробела
*	*	$n$ пробелов

13. Дано натуральное число  $n$ . Разработать рекурсивную функцию для вывода на экран следующей картинки:

1	1 раз
222	3 раза
33333	5 раз
...	( $n$ раз)
33333	(5 раз)
222	(3 раза)
1	(1 раз)

14. Разработать рекурсивную функцию для вывода на экран следующей картинки:

AAAAAAA...AAAAAAA	80 раз
BBBBBBBB...BBBBBBB	78 раз
CCCCCCCC ...CCCCCCC	76 раз
...	...
YYY...YYY	32 раза
ZZ...ZZ	30 раз
YYY...YYY	32 раза
...	...
CCCCCCCC ...CCCCCCC	76 раз
BBBBBBBB...BBBBBBB	78 раз
AAAAAAA...AAAAAAA	80 раз

15. Разработать рекурсивную функцию, которая удаляет из заданной строки все точки.

16. Разработать рекурсивную функцию, которая после каждого вхождения символа  $a$  в строку  $s$  добавляет символ  $b$ .

17. Разработать рекурсивную функцию, которая в заданной строке заменяет все слова, начинающиеся с заглавной буквы, на многоточия.

18. Разработать рекурсивную функцию, которая каждый отрицательный элемент одномерного массива заменяет противоположным по значению элементом.

19. Разработать рекурсивную функцию, которая каждый четный элемент двумерного массива заменяет нулем.

20. Разработать рекурсивную функцию для нахождения максимального элемента и его номера в одномерном массиве.

**IV. Используя механизм перегрузки функций, разработайте две версии функции  $F$ , заголовки которых выглядят следующим образом:**

- 1) float  $F(\text{float } x);$
- 2) void  $F(\text{float } x, \text{ float } &y);$

Продемонстрируйте работу данных функций на примерах.

Функции  $y = f(x)$  использовать из параграфа «Упражнения» (VI) к гл. 3.

**V. Использование функций-шаблонов: для работы с двумерными массивами арифметических типов данных разработать шаблоны ввода и вывода массива, а также шаблон для решения основной задачи.**

1. Заменить все положительные элементы противоположными им числами.
2. Заменить все элементы, меньшие заданного числа, этим числом.
3. Заменить все элементы, попадающие в интервал  $[a, b]$ , нулем.
4. Все элементы, меньшие заданного числа, увеличить в два раза.
5. Подсчитать среднее арифметическое элементов.
6. Подсчитать среднее арифметическое отрицательных элементов.
7. Подсчитать сумму элементов, попадающих в заданный интервал.
8. Подсчитать количество элементов, не попадающих в заданный интервал.
9. Подсчитать количество максимальных элементов.

10. Заменить все минимальные элементы противоположными по значению.

11. Подсчитать среднее арифметическое элементов, расположенных выше главной диагонали.

12. Подсчитать сумму элементов, расположенных на побочной диагонали.

13. Подсчитать среднее арифметическое ненулевых элементов, расположенных над побочной диагональю.

14. Подсчитать среднее арифметическое элементов, расположенных под побочной диагональю.

15. Поменять местами столбцы по правилу: первый с последним, второй с предпоследним и т.д.

16. Если количество строк в массиве четное, то поменять строки местами по правилу: первую строку со второй, третью — с четвертой и т.д. Если количество строк в массиве нечетное, то оставить массив без изменений.

17. Подсчитать норму матрицы по формуле  $\|A\| = \sum_i \max_j a_{i,j}.$

18. Подсчитать норму матрицы по формуле  $\|A\| = \sum_j \max_i a_{i,j}.$

19. Вывести элементы матрицы в следующем порядке:

---

---

---

---

20. Выяснить, является ли матрица симметричной относительно главной диагонали.

### Замечание

Продемонстрировать использование шаблонов на нескольких примерах.

## Самостоятельная работа

**Задача 1.** Разработать рекурсивную функцию для вывода на экран всех возможных разложений натурального числа  $n$  на множители (без повторений). Например, для  $n = 12$  на экран может быть выведено:

```
2 * 2 * 3 = 12
2 * 6 = 12
3 * 4 = 12
```

**Задача 2.** Разработать рекурсивную функцию для вывода на экран всех возможных разложений натурального числа  $n$  на слагаемые (без повторений). Например, для  $n = 5$  на экран может быть выведено:

```
1 + 1 + 1 + 1 + 1 = 5
1 + 1 + 1 + 2 = 5
1 + 1 + 3 = 5
1 + 4 = 5
2 + 1 + 2 = 5
2 + 3 = 5
```

**Задача 3.** Разработать рекурсивную функцию для вычисления определителя заданной матрицы, пользуясь формулой разложения по первой строке

$$\det(A) = \sum_{k=1}^n (-1)^{k+1} a_{1k} \det(B_k).$$

Продемонстрируйте работу данной функции на примерах.

# Глава 9

## ОРГАНИЗАЦИЯ ФАЙЛОВОГО ВВОДА/ВЫВОДА

Мы уже знаем, что в языке C++ механизм ввода-вывода функционирует с помощью **потоков**. Поток — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Чтение данных из потока называется *извлечением*, вывод в поток — *помещением* или *включением*. По направлению обмена потоки можно разделить на *входные* (данные помещаются в поток), *выходные* (извлекаются из потока) и *двунаправленные* потоки (допускается как извлечение, так и включение).

По виду устройств, с которым работает поток, потоки можно разделить на *стандартные, файловые и строковые*.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея. До сих пор ввод-вывод мы осуществляли с помощью стандартных потоков.

Файловые потоки предназначены для обмена информацией с файлами на внешних носителях. Организация файловых потоков будет рассмотрена в данном параграфе.

Строчные потоки позволяют считывать и записывать информацию из областей оперативной памяти. В данном пособии они не рассматриваются.

Потоки C++ обеспечивают надежную работу как со стандартными, так и с определенными пользователями типами данных, а также единообразный и понятный синтаксис.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — *ios* и *streambuf*. Класс *ios* содержит общие для ввода и вывода поля и функции. Класс *streambuf* обеспечивает взаимодействие потоков с физическими устройствами.

### Замечание

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен данных (оперативной памяти, файла на диске, клавиатуры, принтера или экрана). Обмен с потоком для увеличения скорости передачи данных производится через специальную область оперативной памяти — буфер. Фактическая передача данных выполняется при выводе после заполнения буфера, при вводе — если буфер исчерпан.

От этих классов наследуется класс *istream* для входных потоков и *ostream* — для выходных потоков. Два последних класса являются базовыми для класса *iostream*, реализующего двунаправленные потоки. Напомним, что именно класс *iostream* мы использовали для организации стандартного ввода/вывода данных.

В C++ реализованы три класса для работы с файлами: *ifstream* — класс входных файловых потоков; *ofstream* — класс выходных файловых потоков; *fstream* — класс двунаправленных файловых потоков. Эти классы являются производными от классов *istream*, *ostream* и *iostream* соответственно, поэтому они наследуют перегруженные операции << и >>, а также манипуляторы для управления форматированием данных.

Рассмотрим классы для работы с файлами более подробно.

## 9.1. Файловые потоки

Файл — это поименованная совокупность данных на внешнем носителе информации, например на жестком диске. Логически файл можно представить как конечное количество последовательных байтов. Физически файл начинается с некоторого байта памяти. При открытии файла его внутренний указатель устанавливается на начальный байт файла, который имеет индекс «0». Каждый следующий байт файла имеет индекс на единицу больше. Чтение/запись данных производится в текущую позицию указателя. Индекс последнего байта файла определяет его размер в байтах.

По способу доступа файлы можно разделить на последовательные, чтение и запись в которых производится с начала файла байт за байтом, и файлы с произвольным доступом, допускающие чтение и запись в указанную позицию. По внутреннему представлению данных файлы бывают текстовые и двоичные.

В текстовых файлах все данные сохраняются в виде символов, даже числа. При считывании данных из потока (или записи данных в поток) происходят необходимые преобразования типов. Например, при чтении данных строки могут преобразовываться к арифметическим типам, а при записи, наоборот, арифметические типы преобразуются в строковый тип, на что требуются дополнительные временные затраты. Однако текстовые файлы просты для чтения пользователем, а для их создания и редактирования можно пользоваться текстовым редактором, например «Блокнотом».

В двоичных файлах данные сохраняются во внутреннем (машинном) представлении, поэтому они обычно занимают меньше места, и работа с ним и происходит быстрее, так как преобразование типов не производится. Однако просмотр и редактирование двоичных файлов возможны только программным путем.

Работу с файлами через потоки можно осуществить, подключив к программе заголовочный файл "fstream". В общем случае, при про-

граммном использовании файлов предполагаются следующие последовательные этапы: создание потока нужного типа; открытие потока и связывание с ним файла; обмен данных (ввод-вывод); закрытие файла. Рассмотрим данные этапы более подробно.

**Работа с текстовыми файлами.** Напомним, что существуют одонаправленные потоки (потоки ввода и потоки вывода) и двунаправленные потоки. Чтобы открыть входной поток, необходимо объявить потоковый объект типа *ifstream*. Для открытия выходного потока нужно объявить потоковый объект типа *ofstream*. Поток, который предполагается использовать для операций как ввода, так и вывода, должен быть объявлен как *fstream*. Например, при выполнении следующего фрагмента кода будет создан входной поток, выходной поток и поток, позволяющий выполнять операции в обоих направлениях:

```
ifstream in; // входной поток in  
ofstream out; // выходной поток out  
fstream both; // поток ввода-вывода both
```

Создав поток, его нужно связать с файлом. Это можно сделать с помощью функции *open*. Например:

```
in.open("infile.txt");
```

открывает файл по имени *infile.txt*, расположенный в текущем каталоге, и связывает его с входным потоком *in*.

Можно одновременно выполнить две операции — открыть файл и связать его с потоком. Например:

```
// открываем входной поток in  
// и связываем его с файлом infile.txt  
ifstream in ("infile.txt");  
// открываем выходной поток out  
// и связываем его с файлом outfile.txt  
ofstream out ("outfile.txt");  
// открываем двунаправленный поток both  
// и связываем с файлом bothfile.txt  
fstream both ("bothfile.txt");
```

Открыв файл, можно проверить, не произошло ли при этом ошибки. Например:

```
if (!in) // если объект in не готов к применению  
{  
    cout << error; // вывести сообщение об ошибке  
    return -1; // вернуть значение -1, которое сигнализирует  
              // об ошибке  
}
```

Считывать данные из файла можно с помощью операции *>>*, а записывать — с помощью операции *<<*. Например:

```
in >> x; // считываем значение из потока in в переменную x  
out << x; // помещаем значение переменной x в поток out
```

Заметим, что при использовании операции `>>` для считывания данных из текстовых файлов извлечение из потока прекращается при встрече пробельного символа (пробел, табуляция, конец строки), причем сами пробельные символы пропускаются. Чтобы избежать этого, необходимо работать с файлами в двоичном режиме доступа.

Очень часто при считывании данных из файла их количество неизвестно, но требуется считать и обработать все данные. Например, нам необходимо подсчитать среднее арифметическое всех элементов файла, в котором записаны числа. Для этого нужно считывать числа по одному до тех пор, пока в файле не останется ни одного непрочитанного числа. Предположим, что с файлом, содержащим набор чисел, соединен поток `in`. Тогда алгоритм вычисления среднего арифметического всех чисел из файла может быть таким:

```
double next, sum = 0;
int count = 0;
while (in >> next) // 1
{
    sum = sum + next;
    count++;
}
double sr = sum/count;
```

В этом цикле выражение `in >> next` (строка 1) служит одновременно и для считывания очередного числа из потока `in`, и для проверки условия окончания цикла `while`. Таким образом, данное выражение является и оператором, выполняющим некоторое действие, и логическим выражением. Как оператор, оно считывает число из входного потока, а как логическое выражение — возвращает значение `true` или `false`. Если в потоке имеется еще одно число, оно считывается, и логическое выражение оказывается истинным, в результате чего тело цикла выполняется еще один раз. Если же чисел больше не осталось, ничего не вводится, и логическое выражение оказывается ложным, в результате чего цикл завершается. В этом примере переменная `next` имеет тип `double`, но такой метод проверки конца файла одинаково действует и для других типов данных, таких как `int` или `char`.

Для завершения работы с потоком его необходимо закрыть. Это можно сделать с помощью функции `close()`. Например:

```
in.close(); // закрывает входной поток,
            // связанный с файлом infile.txt
out.close(); // закрывает выходной поток,
            // связанный с файлом outfile.txt
```

**Работа с двоичными файлами.** Для открытия двоичных файлов необходимо открыть входной поток, используя спецификатор режима `ios::binary`. Например:

```
ifstream in ("infile.dat", ios::binary);
```

### Замечание

Функции обработки двоичных файлов могут применяться и для работы с файлами, открытыми в текстовом режиме доступа, но при этом могут иметь место преобразования символов, которые сводят на нет основную цель выполнения двоичных файловых операций.

На нижнем уровне двоичного ввода/вывода находятся функции *get* и *put*.

Функция *get* считывает один символ (один байт) из соответствующего потока и помещает его значение в переменную *ch* типа **char**, при этом возвращает ссылку на поток, связанный с предварительно открытым файлом. При считывании символа конца файла данная функция возвратит вызывающему потоку значение *false*. Обращение к функции выглядит следующим образом:

```
in.get(ch);
```

Функция *put* записывает символ *ch* в поток и возвращает ссылку на этот поток. Обращение к функции выглядит следующим образом:

```
in.put(ch);
```

Рассмотрим следующий фрагмент программы:

```
char next;  
in.get(next);
```

Если считываемым символом является пробел, приведенный код не пропускает его, а прочитывает и присваивает переменной *next* значение, равное символу пробела. Если следующим символом является символ перевода строки '*\n*', который означает, что программа достигла конца вводимой строки, вызов *in.get(next)* присваивает переменной *next* значение '*\n*'.

### Замечание

Напомним, что хотя символ перевода строки записывается как пара символов '*\n*', в C++ они интерпретируются как один управляющий символ.

Мы уже сталкивались с вопросом, как считать все данные до конца файла. Для двоичных файлов можно использовать функцию *eof*.

### Замечание

Название функции *eof* представляет собой сокращение от англ. *end of file* — конец файла.

В конце каждого файла имеется специальный маркер конца файла. У функции *eof* нет аргументов, и она возвращает значение *true*, если

прочитан маркер конца файла, *false* — в противном случае. Обратиться к функции можно следующим образом: *in.eof()*.

Рассмотрим в качестве примера следующий оператор:

```
if (!in.eof()) cout << "Not end.";  
else cout << "End of the file.";
```

Логическое выражение после ключевого слова **if** означает «*не достигнут конец файла, связанного с потоком in*». Поэтому если программа еще не достигла конца файла, связанного с потоком *in*, то приведенный оператор выведет на экран следующее: *Not end*. Если же программа достигла конца файла, этот оператор выведет: *End of the file*.

Тогда содержимое файла может быть выведено на экран с помощью следующего цикла **while**:

```
in.get(next);  
while (!in.eof())  
{  
    cout << next;  
    in.get(next);  
}
```

Этот цикл **while** считывает в переменную *next* типа **char** побайтно данные из файла и выводит их на экран. Когда достигается конец файла, значение выражения *in.eof()* становится равным *true*. Поэтому выражение *(!in.eof())* принимает значение *false*, и цикл завершается. Предположим, что файл содержит следующий текст:

```
ab  
c
```

На самом деле этот файл содержит четыре символа:

```
ab'\n'c
```

Приведенный выше цикл прочитает из файлового потока и выведет на экран символ *a*, затем — символ *b*, далее прочитает и выведет на экран символ перевода строки '*\n*', а потом — символ *c*. К этому моменту цикл прочитает все имеющиеся в файле символы, но выражение *in.eof()* по-прежнему будет возвращать *false*. На следующем шаге программа прочитает маркер конца файла, и цикл завершит свою работу. Вот почему в нашем цикле функция *in.get(next)* стоит в конце.

Рассмотренный фрагмент программы можно упростить, используя функцию *peek*. Функция *peek* возвращает следующий символ потока, или значение EOF, если достигнут конец файла.

### Замечание

Напомним, что в C++ прописные и строчные буквы в имени идентификатора различаются. Поэтому *eof* и *EOF* — это два совершенно разных идентификатора: *eof* — это имя функции, которая определяет, достигнут ли конец файла, а *EOF* — это константа, в которой хранится значение маркера конца файла.

Тогда содержимое файла может быть выведено на экран с помощью следующего цикла:

```
while (in.peek() != EOF)
{
    in.get(next);
    cout << next;
}
```

Функции *get* и *put* используются для считывания данных побайтно, что позволяет нам обрабатывать текстовые данные, представленные в ASCII-кодировке (один символ — один байт). Однако текстовые данные могут представляться и в кодировке Unicode (один символ — два байта). Кроме того, в двоичных файлах могут храниться последовательности целых и вещественных типов, где каждое значение в машинном представлении занимает от 2 до 10 байт, а также пользовательские типы, размеры которых заранее неизвестны. В этом случае использование функций *get* и *put* невозможно.

Для считывания и записи блоков двоичных данных используются функции *read()* и *write()*, которые также являются функциями-членами потоковых классов соответственно для ввода и для вывода. Прототипы данных функций выглядят следующим образом:

```
istream &read(char *<буфер>, <число_байт>);
ostream &write(const char *<буфер>, <число_байт>);
```

Функция *read()* считывает из вызывающего потока столько байт, сколько задано параметром *<число\_байт>*, и передает их в *<буфер>*. Если конец файла достигнут до того, как было считано нужное количество байт, то выполнение функции *read()* прекращается, а в буфере оказывается столько байт, сколько их было в файле. Узнать, сколько символов было считано, можно, обратившись к входному потоку с помощью функции-члена *gcount()*. Обратиться к функции *read()* для открытого входного потока *in* можно следующим образом:

```
// Вместо double может быть указан любой другой тип данных
double i;
in.read((char*)&i, sizeof(double));
```

Функция *write()* записывает в соответствующий поток из *<буфер>* столько байт, сколько задано параметром *<число\_байт>*. Обратиться к функции *write()* для открытого выходного потока *out* можно следующим образом:

```
// Вместо int может быть указан любой другой тип данных
int i;
out.write((char*)&i, sizeof(int));
```

Рассмотрим следующую программу:

```
#include "fstream"
using namespace std;
```

```

int main()
{
    // работа с текстовым файлом
    ofstream out("outfile.txt");
    for (double i = 0; i < 10; i += 0.5)
        out << i << ' '; // запись данных в текстовый файл
    out.close();
    // работа с двоичным файлом
    out.open("outfile.dat", ios::binary);
    for (double i = 0; i < 10; i += 0.5)
        // запись данных в двоичный файл
        out.write((char *)&i, sizeof(double));
    out.close();
}

```

Если открыть созданные файлы в текстовом редакторе, то мы увидим следующие данные:

#### **file.txt**

```
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5
```

#### **file.dat**

000000000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E0 3F	.....?.....?
000000010	00 00 00 00 00 00 F0 3F 00 00 00 00 00 00 F8 3F	.....?.....?
000000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 40	.....@.....@
000000030	00 00 00 00 00 00 00 00 00 08 40 00 00 00 00 00 00 OC 40	.....@.....@
000000040	00 00 00 00 00 00 10 40 00 00 00 00 00 00 12 40	.....@.....@
000000050	00 00 00 00 00 00 14 40 00 00 00 00 00 00 16 40	.....@.....@
000000060	00 00 00 00 00 00 18 40 00 00 00 00 00 00 1A 40	.....@.....@
000000070	00 00 00 00 00 00 1C 40 00 00 00 00 00 00 1E 40	.....@.....@
000000080	00 00 00 00 00 00 20 40 00 00 00 00 00 00 21 40	.....@.....!@
000000090	00 00 00 00 00 00 22 40 00 00 00 00 00 00 23 40	....."@.....#@
0000000a0		

Таким образом, в текстовом файле отображаются сами числа, а в двоичном — их машинное представление. Как видим, машинное представление не читаемо для пользователя. Прочитать данные из бинарного файла можно только программным путем. Например, следующим образом:

```

#include "iostream"
#include "fstream"
using namespace std;
int main()
{
    // файл file.dat должен существовать
    ifstream in("file.dat", ios::binary);
    double i;
    // чтение данных из файла
    while (in.read((char *)&i, sizeof(double)))
        cout << i << ' ';
    in.close();
}

```

#### **Замечание**

Обратите внимание на то, что чтение данных до конца файла осуществляется аналогично тому, как производилось чтение в текстовом файле. Только

вместо оператора `in >> i` для двоичного файла в условии завершения стоит оператор чтения:

```
in.read((char*) &i, sizeof(double))
```

---

**Произвольный доступ.** До сих пор мы работали с файлами в режиме последовательного доступа. Рассмотрим теперь, как организовать доступ к произвольной компоненте файла.

#### Замечание

---

Произвольный доступ имеет смысл только для бинарных файлов, в которых можно точно определить размер компонентов файла в байтах. В текстовых файлах в общем случае длина каждого компонента может быть различной.

---

Для обеспечения произвольного доступа используется внутренний указатель файла, который определяет текущий индекс обрабатываемого байта файла (напомним, что байты файла нумеруются, как и элементы массива, с нуля), и две пары функций. Одна пара функций, `seekg()`, `seekp()`, используется потоками ввода и устанавливает указатель в нужную позицию, а вторая пара функций, `tellp()`, `tellg()`, используется потоками вывода и сообщает текущую позицию указателя.

#### Замечание

---

Фактически это две функции — `seek` и `tell`, каждая из которых используется с двумя разными суффиксами: *g* (*getting*) означает получение данных, *p* (*putting*) — помещение (запись) данных.

---

Функция `seekg(n)` устанавливает указатель в позицию с номером *n* внутри файла для чтения. Обращение к ней для открытого входного потока `in` может выглядеть следующим образом:

```
// указатель будет установлен на позицию с номером 3  
in.seekg(3);
```

Функция `seekp(n)` устанавливает указатель на позицию с номером *n* внутри файла для записи. Обращение к ней для открытого выходного потока `out` может выглядеть следующим образом:

```
// указатель будет установлен на позицию с номером 3  
out.seekp(3);
```

С помощью констант позиционирования можно уточнить позицию указателя в обрабатываемом потоке:

- `ios::beg` — позиционирование относительно начала потока;
- `ios::cur` — позиционирование относительно текущей позиции в потоке;
- `ios::end` — позиционирование относительно конца потока.

Например:

```
// Установить указатель на позицию с номером 3
// относительно начала файла
in.seekg(3, ios::beg);
// Установить указатель на позицию с номером 3
// относительно конца файла
in.seekg(3, ios::end);
// Перемещаем указатель на 3 индекса относительно текущей позиции
in.seekg(3, ios::cur);
```

#### Замечание

Константы позиционирования *beg*, *cur* перемещают указатель в направлении от начала к концу файла, а константа *end* — от конца к началу файла.

Функция *tellg()* возвращает текущую позицию указателя потока ввода. Обращение к ней для открытого входного потока *in* может выглядеть следующим образом:

```
// в переменную n будет записана текущая позиция чтения
streampos n = in.tellg();
```

Функция *tellp()* возвращает текущую позицию маркера потока вывода. Обращение к ней для открытого выходного потока *out* может выглядеть следующим образом:

```
// в переменную n будет записана текущая позиция записи
streampos n = in.tellp();
```

#### Замечание

*streampos* — специальный тип данных, который используется для хранения индекса указателя в файле. Определен в заголовочном файле "fstream".

## 9.2. Примеры решения задач с использованием файлового ввода/вывода

1. Дан текстовый файл *f.txt*. Переписать в файл *g.txt* все его строки в перевернутом виде.

#### Замечание

Напоминаем, что текстовый файл должен быть предварительно создан в текстовом редакторе, например «Блокноте».

```
#include "fstream"
#include "iostream"
```

```

#include "string"
using namespace std;
int main()
{
    ifstream in("f.txt");
    ofstream out("g.txt");
    string s;
    // пока не прочитан маркер конца файла
    while (in.peek() != EOF)
    {
        // читаем очередную строку из файла f.txt
        getline(in, s);
        for (unsigned int i = 0; i < s.length() / 2;
             i++) // зеркально отображаем строку
        {
            char a = s[i];
            s[i] = s[s.length() - 1 - i];
            s[s.length() - 1 - i] = a;
        }
        // записываем измененную строку в файл g.txt
        out << s << endl;
    }
    in.close();
    out.close();
    return 0;
}

```

Результат работы программы:

*f.txt*

Многие вещи нам непонятны не потому,  
что наши понятия слабы; но потому, что сии  
вещи не входят в кругозор наших понятий.  
Козьма Прутков,  
Плоды раздумья, мысль 66

*g.txt*

,умотоп ен ынтянопен ман ищев еигонM  
иис отч ,умотоп он ;ыбалс яитяноп ишан отч  
.ийтияноп хишан розогурк в тядохв ен ищев  
,воктурП амъзоК  
66 ылсым ,ямъудзар ыдолП

**2.** Дан файл *f.txt*, компонентами которого являются целые числа. В файл *g.txt* переписать все неотрицательные компоненты файла *f.txt*, кратные трем.

```

#include "fstream"
#include "iomanip"
using namespace std;
int main()
{
    ifstream in("f.txt");
    ofstream out("g.txt");
    int i;
    // пока не прочитан маркер конца файла, читаем из потока
    // очередной компонент
    while (in >> i)
    {
        // если компонент отвечает заданным требованиям,
        if ((i >= 0) &&
            (i % 3 == 0))

```

```
        out << setw(5) << i; // то помещаем его в выходной поток
    }
in.close();
out.close();
return 0;
}
```

Результат работы программы:

```
f.txt:
-9 -8
-7 6 7
8
9

g.txt:
6     9
```

#### Замечание

При формировании выходного потока мы использовали манипулятор для форматированного вывода `setw`. Вспомните, что делает данный манипулятор, и подумайте, что будет, если его не использовать.

3. Дан файл *f.txt*, в котором записан текст. Переписать этот текст в файл *g.txt* с сохранением форматирования, заменив при этом все вхождения символа С на С++.

#### Замечание

При работе с текстовым файлом мы использовали бинарную функцию `get()` для считывания файла посимвольно.

```
#include "fstream"
#include "iostream"
using namespace std;
int main()
{
    char symbol;
    ifstream in("f.txt");
    ofstream out("g.txt");
    while (in.peek() != EOF)
    {
        // читаем очередной символ из файла
        in.get(symbol);
        // если он равен символу С
        if (symbol == 'C')
            // то вместо него в выходной поток помещаем строку C++
            out << "C++";
        // иначе в выходной поток помещаем текущий символ
        else
```

```

        out << symbol;
    }
in.close();
out.close();
return 0;
}

```

Результат работы программы:

f.txt:  
Abc CBA C+c

g.txt:  
Abc C++BA C++c

4. Создать бинарный файл *f.dat*, записав в него вещественные числа из интервала от *a* до *b* с шагом *h*. Вывести на экран компоненты файла *f.dat* через одну, начиная со второй:

#### Замечание

---

Напоминаем, что двоичные файлы создаются и просматриваются только программным путем.

---

```

#include "iostream"
#include "fstream"
using namespace std;
int main()
{
    ofstream out("f.dat", ios::binary);
    double a, b, h, i;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "h = ";
    cin >> h;
    // записываем данные в двоичный файл
    for (i = a; i <= b; i += h)
        out.write((char *)&i, sizeof(i));
    out.close();
    ifstream in("f.dat", ios::binary);
    // смещаем указатель относительно начала файла на столько
    // байтов, сколько отводится для хранения вещественного
    // числа, тем самым перемещаем указатель на второе
    // вещественное число в файле
    in.seekg(sizeof(double));
    while (in.peek() != EOF) {
        // читаем данные из двоичного файла
        in.read((char *)&i, sizeof(double));
        cout << i << ' ';
    }
}

```

```

// смещаем указатель относительно текущей позиции
// указателя на столько байтов, сколько отводится
// для хранения вещественного числа, тем самым
// пропускаем одно вещественное число в файле
in.seekg(sizeof(double), ios::cur);
}
in.close();
return 0;
}

```

Результат работы программы:

Входные данные:

```

a = 0
b = 10
h = 0.5

```

Выходные данные:

```

0.5   1.5   2.5   3.5   4.5   6.5   7.5   8.5   9.5

```

## Упражнения

### Замечание

Двоичные файлы создаются программным путем. Тестовые файлы нужно предварительно создать в текстовом редакторе, например «Блокноте».

#### I. Работа с одним текстовым файлом.

1. Найти количество строк, которые начинаются с данной буквы.
2. Найти количество строк, которые начинаются и заканчиваются одной буквой.
3. Найти самую длинную строку и ее длину.
4. Найти самую короткую строку и ее длину.
5. Найти номер самой длинной строки.
6. Найти номер самой короткой строки.
7. Выяснить, имеется ли в файле строка, которая начинается с данной буквы.

Если да, то напечатать ее.

8. Напечатать первый символ каждой строки.
9. Напечатать символы с  $k_1$  по  $k_2$  в каждой строке.
10. Напечатать все нечетные строки.
11. Напечатать все строки, в которых имеется хотя бы один пробел.
12. Напечатать все строки, длина которых равна данному числу.
13. Напечатать все строки, длина которых меньше заданного числа.
14. Напечатать все строки с номерами от  $k_1$  до  $k_2$ .
15. Получить слово, образованное  $k$ -ми символами каждой строки.
16. Переписать в новый файл все строки, вставив в конец каждой строки ее номер.
17. Переписать в новый файл все строки, вставив в конец каждой строки количество символов в ней.
18. Переписать в новый файл все строки, длина которых больше заданного числа.
19. Переписать в новый файл все строки четной длины.

20. Переписать в новый файл все строки, удалив из них символы, стоящие на четных местах.

## II. Работа с несколькими текстовыми файлами.

1. Дан файл  $f$ , компонентами которого являются целые числа. Переписать все четные числа в файл  $g$ , нечетные — в файл  $h$ .

2. Дан файл  $f$ , компонентами которого являются целые числа. Переписать все отрицательные числа в файл  $g$ , положительные — в файл  $h$ .

3. Данны два файла с числами. Поменять местами их содержимое (использовать вспомогательный файл).

4. Данны два файла с числами. Получить новый файл, каждый элемент которого равен сумме соответствующих компонентов заданных файлов (количество компонентов в исходных файлах одинаковое).

5. Данны два файла с числами. Получить новый файл, каждый компонент которого равен наибольшему из соответствующих компонентов заданных файлов (количество компонентов в исходных файлах одинаковое).

6. Данны два файла с числами. Получить новый файл, каждый компонент которого равен среднему арифметическому значению соответствующих компонентов заданных файлов (количество компонентов в исходных файлах одинаковое).

7. Данны два файла с числами. Получить новый файл, записав в него сначала все положительные числа из первого файла, потом все отрицательные числа из второго.

8. Данны два файла с числами. Получить новый файл, записав в него сначала все четные числа из первого файла, потом все нечетные числа из второго.

9. Данны два файла с числами. Получить новый файл, в котором на четных местах будут стоять компоненты, которые стоят на четных местах в первом файле, а на нечетных — компоненты, которые стоят на нечетных во втором (количество компонентов в исходных файлах одинаковое).

10. Дан файл  $f$ , компонентами которого являются символы. Переписать в файл  $g$  все знаки препинания файла  $f$ , а в файл  $h$  — все остальные символы файла  $f$ .

11. Дан файл  $f$ , элементами которого являются символы. Переписать в файл  $g$  все цифры файла  $f$ , а в файл  $h$  — все остальные символы файла  $f$ .

12. Данны два файла с одинаковым количеством компонентов, компонентами которых являются символы. Выяснить, совпадают ли попарно их компоненты. Если нет, получить номер первого элемента, в котором эти файлы отличаются.

13. Дан файл, компонентами которого являются целые числа. Переписать в новый файл сначала все отрицательные компоненты из первого, потом все положительные.

14. Дан файл, компонентами которого являются символы. Создать новый файл таким образом, чтобы на четных местах у него стояли компоненты, стоящие на нечетных в первом файле, и наоборот.

15. Дан файл, компонентами которого являются числа. Число компонентов файла делится на два. Создать новый файл, в который будет записываться наименьшее из каждой пары чисел первого файла.

16. Дан файл, компонентами которого являются числа. Число компонентов файла делится на два. Создать новый файл, в который будет записываться среднее арифметическое из каждой пары чисел первого файла.

17. Дан файл, компонентами которого являются символы. Переписать все символы в новый файл в обратном порядке.

18. Данны два файла с одинаковым количеством компонентов, компонентами которых являются натуральные числа. Создать новый файл, в который будут

записываться числа по следующему правилу. Берется первое число из первого файла и первое из второго. Если одно из них делится нацело на другое, то их частное записывается в новый файл. Затем берется второе число из первого файла и второе число из второго и т.д.

19. Дан файл, компонентами которого являются символы. Переписать в новый файл все символы, которым в первом файле предшествует данная буква.

20. Дан файл, компонентами которого являются символы. Переписать в новый файл все символы, за которыми в первом файле следует данная буква.

### III. Работа с двоичными файлами.

1. Создать файл и записать в него квадраты натуральных чисел от 1 до  $n$ . Вывести на экран все компоненты файла с нечетным порядковым номером.

2. Создать файл и записать в него степени числа 3. Вывести на экран все компоненты файла с четным порядковым номером.

3. Создать файл и записать в него обратные натуральные числа  $1, \frac{1}{2}, \dots, \frac{1}{n}$ .

Вывести на экран все компоненты файла с порядковым номером, кратным трем.

4. Создать файл и записать в него  $n$  первых членов последовательности Фибоначчи. Вывести на экран все компоненты файла с порядковым номером, не кратным трем.

5. Создать файл, состоящий из  $n$  целых чисел. Вывести на экран все четные числа данного файла.

6. Создать файл, состоящий из  $n$  целых чисел. Вывести на экран все отрицательные числа данного файла.

7. Создать файл, состоящий из  $n$  целых чисел. Вывести на экран все числа данного файла, попадающие в заданный интервал.

8. Создать файл, состоящий из  $n$  целых чисел. Вывести на экран все числа данного файла, не попадающие в заданный интервал.

9. Создать файл, состоящий из  $n$  целых чисел. Вывести на экран все числа данного файла, не кратные заданному числу.

10. Создать файл, состоящий из  $n$  вещественных чисел. Вывести на экран все числа данного файла, не попадающие в данный диапазон.

11. Создать файл, состоящий из  $n$  вещественных чисел. Вывести на экран все числа данного файла с нечетными порядковыми номерами, большие заданного числа.

12. Создать файл, состоящий из  $n$  вещественных чисел. Вывести на экран все числа данного файла с четными порядковыми номерами, меньшие заданного числа.

13. Создать файл, состоящий из  $n$  вещественных чисел. Найти сумму всех положительных чисел данного файла.

14. Создать файл, состоящий из  $n$  вещественных чисел. Подсчитать среднее арифметическое чисел файла, стоящих на четных позициях.

15. Создать файл, состоящий из  $n$  вещественных чисел. Найти максимальное значение среди чисел файла, стоящих на нечетных позициях.

## Самостоятельная работа

**Задача 1.**  $N$  человек играли в карточную игру. После каждой раздачи был один выигравший и один проигравший. Они сыграли  $M$  раздач и каждый раз записывали, кто у кого сколько выиграл (в одной раздаче нельзя выиграть больше 100 очков). Теперь игроки хотят узнать, сколько каждый из них выиграл. Помогите им.

Во входном текстовом файле в первой строке находится натуральное число  $N$  ( $1 \leq N \leq 50$ ). В следующих  $N$  строках записаны имена игроков. Длина имени не превосходит 10 символов и не содержит пробелов. В следующей строке записано целое число  $M$  ( $0 \leq M \leq 100$ ). Далее записаны  $M$  строк с результатами раздач в формате:

<выигравшего игрока> <имя проигравшего игрока> <выигрыш>.

В выходной файл необходимо записать  $N$  строк в формате:

<имя игрока> <выигрыш>

Если игрок проиграл, то его выигрыш отрицательный. Все имена и числа в одной строке разделяются единственным пробелом. Строки располагаются в файле в произвольном порядке. Например:

**input.txt**

```
3
Петя
Вася
Саша
3
Петя Саша 10
Петя Вася 20
Саша Вася 10
```

**output.txt**

```
Петя 30
Саша 0
Вася -30
```

**Задача 2.** В сообщении могут встречаться номера телефонов, записанные в формате xx-xx-xx, xxx-xxx или xxx-xx-xx. Найти все номера телефонов, содержащиеся в сообщении.

Исходное сообщение содержится в текстовом файле *input.txt*. Номера телефонов следует вывести в файл *output.txt*, при этом каждый номер выводится с новой строки. Например:

**input.txt**

У Васи телефон 12-34-56. А с Петей можно связаться по номеру 789-012.

**output.txt**

```
12-34-56
789-012
```

**Задача 3.** В сообщении могут содержаться даты в формате дд.мм.гг. Найти все даты, содержащиеся в сообщении.

Исходное сообщение содержится в текстовом файле *input.txt*. Даты следует вывести в файл *output.txt*, при этом каждая дата выводится с новой строки и через пробел указывается дата предшествующего дня. Например:

**input.txt**

У Васи день рождения 01.01.84. А с Петей я  
познакомился 01.12.95. Встреча друзей  
назначена на 25.07.09.

***output.txt***

```
01.01.84 31.12.83  
01.12.95 30.11.94  
25.07.09 24.07.09
```

**Задача 4.** В сообщении может содержаться время в формате чч:мм:сс. Преобразовать каждое время к формату чч:мм, применив правило округления до целого числа минут, и вывести на экран.

Исходное сообщение содержится в текстовом файле *input.txt*. Время следует вывести в файл *output.txt*, при этом каждое время выводится с новой строки. Например:

***input.txt***

```
Вася потратил 01:24:34 на подготовку к  
контрольной работе. А Петя 01:59:12 и  
лег спать в 23:59:59.
```

***output.txt***

```
01:25  
01:59  
00:00
```

**Задача 5.** В сообщении могут содержаться IP-адреса компьютеров в формате d.d.d.d, где *d* — целое число из диапазона от 0 до 255. Найти все IP-адреса, содержащиеся в сообщении.

Исходное сообщение содержится в текстовом файле *input.txt*. Адреса следует вывести в файл *output.txt*, при этом каждый IP-адрес выводится с новой строки. Например:

***input.txt***

```
У моего компьютера IP-адрес 127.23.3.78.  
А у Пети то ли 127.23.3.258, то ли 127.23.32.58
```

***output.txt***

```
127.23.3.78  
127.23.32.58
```

# Глава 10

## СТРУКТУРЫ

### 10.1. Общие сведения

Структуры, как и массивы, относятся к *составным* типам данных. Однако, в отличие от массива, элементы которого однотипны, структура может содержать элементы разных типов. Синтаксис описания структуры:

```
struct <имя структуры>
{
    <тип 1> <идентификатор 1>;
    <тип 2> <идентификатор 2>;
    ...
    <тип n> <идентификатор n>;
};
```

Ключевое слово **struct** используется для определения типа данных структура. Идентификаторы, объявленные внутри фигурных скобок, называются членами-данными структуры, или полями. Например, рассмотрим следующее описание структуры:

```
struct sotrudnik
{
    string familiya, imya, otchestvo, adres;
    int nomer;
};
```

Здесь определяется структура *sotrudnik*, имеющая пять членов-данных, четыре из которых имеют тип *string* и один — тип *int*.

После определения структуры ее тип может использоваться так же, как и любой из стандартных типов данных **int**, **float** и т.д. Например:

```
sotrudnik sotr; // описание переменной sort типа sotrudnik
// описанием одномерного массива,
// где каждый элемент имеет тип sotrudnik
sotrudnik mas[10];
sotrudnik *p; // описание указателя
```

Можно пропустить один шаг и описать переменную, указатель или массив, где базовым элементом является тип *sotrudnik*, следующим образом:

```

struct sotrudnik
{
    string familiya, imya, otchestvo, adres;
    int nomer;
} sort, *p, mas[10];

```

В данном случае переменная, указатель и массив описываются в момент определения самой структуры.

При описании переменной типа структура можно проводить ее инициализацию, для чего значения ее элементов перечисляются в фигурных скобках в порядке их описания. Например, следующим образом:

```
sotrudnik sort = {"Ivanov", "Ivan", "Ivanovich", "Saratov", "123456"};
```

или

```

struct sotrudnik
{
    string familiya, imya, otchestvo, adres;
    int nomer;
} sort = {"Ivanov", "Ivan", "Ivanovich", "Saratov", "123456"};

```

Доступ к полям структуры выполняется с помощью операций выбора: при обращении к полю через имя переменной используется операция «.» (точка), при обращении через указатель используется операция ->. Например:

```

sotr.familiya = "Ivanov";
mas[0].imya = "Ivan";
p->otchestvo = "Ivanovich";

```

В программе переменные типа структура могут использоваться как переменные любых других типов. Например:

```

// Ввод данных в переменную sort
cin >> sotr.familiya;
cin >> sotr.imya;
cin >> sotr.otchestvo;
cin >> sotr.adres;
cin >> sotr.nomer;
// Вывод данных, связанных с указателем p
cout << p->familiya << '\t' << p->imya << '\t' << p->otchestvo <<
'\t'
    << p->sotr.adres << endl;
// обработка массива
for (int i = 0; i < n; i++)
    if (mas[i].familiya == poisk)
        cout << mas[i].adres;

```

Структуры языку C++ «достались в наследство» от языка С. В C++ определение структуры расширилось за счет включения в нее членовых функций, в том числе конструкторов и деструкторов. Рассмотрим расширенное описание структуры:

```

struct <имя структуры>
{
    // открытые члены – данные и функции
    ...
private:
    // закрытые члены – данные и функции
    ...
};

```

Обратите внимание на введение нового ключевого слова — *private*, которое сообщает компилятору о том, что следующие за ним члены структуры являются закрытыми (доступ к ним возможен только из данной структуры).

Рассмотрим следующий пример:

```

#include "iostream"
#include "cmath"
using namespace std;
struct point // описание структуры
{
    int x, y;      // открытые члены-данные
    void show();   // открытый член-функция
private:
    double dlina(); // закрытый член-функция
};
double point::dlina() // реализация члена-функции dlina
{
    return sqrt((double)(x * x + y * y));
}
void point::show() // реализация члена-функции show
{
    cout << "Координаты точки: (" << x << ", " << y << ")" << endl;
    double d = dlina(); // допустимое обращение к закрытому члену
                        // структуры
    cout << "Расстояние до начала координат: " << d << endl;
    cout << endl;
}
int main()
{
    point a, b; // описание переменных типа point
    a.x = 1;
    a.y = 1; // обращение к открытым членам-данным структуры point
    a.show(); // обращение к открытому члену-функции структуры point
    // Ошибка! Недопустимо обращаться к закрытому члену
    // структуры из произвольного места программы.
    // double z = a.dlina();
    b.x = 4;
    b.y = 3; // обращение к открытым членам-данным структуры point
    b.show(); // обращение к открытому члену-функции структуры point
    return 0;
}

```

Структура может содержать конструктор, который используется для инициализации членов-данных структуры. Добавим в предыдущий пример конструктор:

```

#include "iostream"
#include "cmath"
using namespace std;
struct point // описание структуры
{
    int x, y;           // открытые члены-данные
    point(int a, int b); // конструктор
    void show();        // открытый член-функция
private:
    double dlina(); // закрытый член-функция
};
point::point(int a, int b) // реализация конструктора
{
    x = a;
    y = b;
}
double point::dlina() // реализация члена-функции dlina
{
    return sqrt((double)(x * x + y * y));
}
void point::show() // реализация члена-функции show
{
    cout << "Координаты точки:" << x << ", " << y << endl;
    double d = dlina();
    cout << "Расстояние до начала координат: " << d << endl;
    cout << endl;
}
int main()
{
    point *a = new point(1, 1); // вызов конструктора
    // обращение к открытому члену-функции структуры point
    a->show();
    point *b; // описание указателя на тип point
    // ошибка: так как b - это только указатель на тип point
    // и еще не произвдилось выделение памяти под данный
    // объект, следовательно, невозможно обратиться к его членам
    b->x = 3;
    b->y = 4;
    // устанавливаем указатель на b на область
    // памяти, связанную с указателем a
    b = a;
    b->show();
    b->x = 100;
    a->show();
    b->show();
    return 0;
}

```

Имя конструктора совпадает с именем структуры. Для конструктора не указывается тип возвращаемого значения, так как назначение конструктора — инициализировать члены-данные. В качестве параметров конструктору передаются аргументы, которые будут использоваться при инициализации.

Если структура содержит конструктор, то работа с ней возможна только через указатели. При описании указателя на тип структура можно сразу вызвать конструктор с помощью операции `new`. Например, в нашем примере имеется строка:

```
point *a = new point(1,1);
```

где `*a` — это указатель на тип структура, а после операции `new` мы указываем имя конструктора и в круглых скобках перечисляем значения параметров.

Так как `a` является указателем, то для обращения к полям структуры нужно использовать операцию выбора `->`. Например, чтобы обратиться к полю `x`, мы записываем `a -> x`.

После того как была выполнена операция присваивания `b = a`, `a` и `b` стали указывать на одну и ту же область памяти. Теперь если изменить значение по указателю `b`, то, обратившись к члену данных `show` для `a` и `b`, мы получим одинаковые результаты. Этот факт нужно учитывать при работе с указателями.

#### Замечание

В C++ тип данных структура получил свое дальнейшее развитие — появились классы. Синтаксис класса очень похож на синтаксис структуры, хотя вместо ключевого слова `struct` используется ключевое слово `class`, а также по умолчанию в структуре все члены открыты, а в классе — закрыты. Появление классов дало развитие новой технологии программирования — объектно-ориентированному программированию, реализующему такие механизмы, как наследование, полиморфизм, инкапсуляция. Более подробно классы будут рассмотрены ниже.

## 10.2. Примеры решения задач

1. Окружность в пространстве задана своим центром и радиусом. Определить, какое количество точек заданного в пространстве множества лежит на окружности.

```
#include "iostream"
#include "cmath"
using namespace std;
// структура для хранения координат точки в пространстве
struct point
{
    int x, y, z;
};
// функция, вычисляющая расстояние между
// двумя точками в пространстве
double dlina(point a, point b)
{
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2)
                + pow(a.z - b.z, 2));
```

```

}

int main()
{
    point circle, a[10];
    double r;
    int n;
    cout << "Введите центр окружности: ";
    cin >> circle.x >> circle.y >> circle.z;
    cout << "Введите радиус окружности: ";
    cin >> r;
    cout << "Введите количество точек в множестве: ";
    cin >> n;
    // ввод координат точек заданного множества
    for (int i = 0; i < n; i++)
    {
        cout << "Введите координаты " << i << "-й точки: ";
        cin >> a[i].x >> a[i].y >> a[i].z;
    }
    int k = 0;
    for (int i = 0; i < n; i++)
        // проверяем, лежит ли точка на заданной окружности
        if (dlina(circle, a[i]) == r)
            k++;
    cout << "Количество точек множества, "
        << "лежащих на заданной окружности, = "
        << k;
    return 0;
}

```

**2.** Дан файл *input.txt*, в котором содержится библиотечная ведомость. Каждая строка ведомости содержит следующую информацию: библиотечный номер, фамилия автора, название книги, год издания. В файл *output.txt* вывести данную библиотечную ведомость, удалив из нее книгу с заданным номером.

```

#include <iostream>
#include <iomanip>
using namespace std;
// открываем глобальные файловые потоки
ifstream in("input.txt");
ofstream out("output.txt");
// описание структуры
struct bibl
{
    int number, year;
    char family[20], nazv[50];
    void show(); // член-функция для вывода информации на экран
    void print(); // член-функция для вывода информации в файл
};

void bibl::show()
{
    cout << setw(8) << number << setw(10) << family << setw(8)

```

```

        << nazv << setw(8) << year << endl;
    }
void bibl::print()
{
    out << setw(8) << number << setw(10) << family << setw(8)
        << nazv << setw(8) << year << endl;
}
int main()
{
    bibl book[10]; // описываем массив структур
    int num;
    int i, j, n = 0;
    if (!in)
        cout << "Ошибка при открытии файла input.txt\n";
    else
    {
        while (in.peek() != EOF) // чтение данных из файла
        {
            in >> book[n].number;
            in >> book[n].family;
            in >> book[n].nazv;
            in >> book[n].year;
            // вывод прочитенных данных на экран через член-функцию
            book[n].show();
            n++;
        };
        cout << "Введите номер книги, которую\n";
        cout << "нужно убрать из списка:";
        cin >> num;
        if (!out)
            cout << "Ошибка при открытии файла output.txt\n";
        else
        {
            j = 0;
            for (i = 0; i < n && !j; i++)
                // поиск структуры по совпадению поля
                // number с num для удаления
                if (book[i].number == num)
                {
                    // выполняем сдвиг в массиве на одну позицию вправо
                    for (j = i; j < n - 1; j++)
                        book[j] = book[j + 1];
                    n--; // уменьшаем количество книг на 1 после удаления
                }
            // записываем результат в новый файл
            for (i = 0; i < n; i++)
                // вывод данных в файл через член-функцию структуры
                book[i].print();
        }
    }
    in.close();
    out.close(); // закрываем потоки
    return 0;
}

```

Результат работы программы:

***input.txt***

```
1 Ivanov C++ 2008
2 Petrov Java 2007
3 Petrov C/C++ 2008
4 Ivanov C# 2009
номер книги
2
```

***output.txt***

```
1    Ivanov      C++      2008
3    Petrov      C/C++    2008
4    Ivanov      C#       2009
```

## Упражнения

I. Решить задачу, используя структуру *point* для хранения координат точки:

**Замечание**

В задачах с четными номерами множество точек задано на плоскости, в задачах с нечетными номерами множество точек задано в пространстве.

- 1—2. Найти точку, которая наиболее удалена от начала координат.
  - 3—4. Найти точку, которая наименее удалена от начала координат.
  - 5—6. Найти две наиболее удаленных друг от друга точки.
  - 7—8. Найти две наиболее близко расположенных друг к другу точки.
  - 9—10. Найти такую точку, что шар радиуса  $R$  с центром в этой точке содержит максимальное число точек заданного множества.
  - 11—12. Найти такую точку, что шар радиуса  $R$  с центром в этой точке содержит минимальное число точек заданного множества.
  - 13—14. Найти такую точку, сумма расстояний от которой до остальных точек множества максимальна.
  - 15—16. Найти такую точку, сумма расстояний от которой до остальных точек множества минимальна.
  - 17—18. Найти три различные точки из заданного множества точек, образующих треугольник наибольшего периметра.
  - 19—20. Найти три различные точки из заданного множества точек, образующих треугольник наименьшего периметра.
- II. Решить задачу, используя структуру, содержащую члены-данные и члены-функции.

**Замечание**

Во всех задачах данного параграфа подразумевается, что исходная информация хранится в текстовом файле *input.txt*, каждая строка которого содержит полную информацию о некотором объекте. Результирующая информация должна быть записана в файл *output.txt*.

1. На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, год рождения, домашний адрес, какую

школу окончил. Вывести в новый файл информацию о студентах, окончивших заданную школу.

2. На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, год рождения, домашний адрес, какую школу окончил. Вывести в новый файл список студентов, удалив из него студентов, окончивших школу в текущем году.

3. На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, номер группы, результаты сдачи трех экзаменов. Вывести в новый файл информацию о студентах, успешно сдавших сессию.

4. На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, номер группы, результаты сдачи трех экзаменов. Вывести в новый файл список студентов, удалив из него информацию о студентах, не сдавших сессию.

5. На основе данных входного файла составить багажную ведомость камеры хранения, включив следующие данные: ФИО пассажира, количество вещей, общий вес вещей. Вывести в новый файл информацию о тех пассажирах, средний вес багажа которых превышает заданный.

6. На основе данных входного файла составить багажную ведомость камеры хранения, включив следующие данные: ФИО пассажира, количество вещей, общий вес вещей. Вывести в новый файл багажную ведомость, удалив из нее информацию о тех пассажирах, средний вес багажа которых меньше заданного.

7. На основе данных входного файла составить автомобильную ведомость, включив следующие данные: марка автомобиля, номер автомобиля, фамилия его владельца, год приобретения, пробег. Вывести в новый файл информацию об автомобилях, выпущенных ранее определенного года.

8. На основе данных входного файла составить автомобильную ведомость, включив следующие данные: марка автомобиля, номер автомобиля, фамилия его владельца, год приобретения, пробег. Вывести в новый файл автомобильную ведомость, удалив из нее информацию об автомобилях, пробег которых менее заданного значения.

9. На основе данных входного файла составить список сотрудников учреждения, включив следующие данные: ФИО, год принятия на работу, должность, заработка плата, рабочий стаж. Вывести в новый файл информацию о сотрудниках, имеющих заработную плату ниже определенного уровня.

10. На основе данных входного файла составить список сотрудников учреждения, включив следующие данные: ФИО, год принятия на работу, должность, заработка плата, рабочий стаж. Вывести в новый файл список сотрудников учреждения, удалив из него информацию о сотрудниках, принятых на работу в текущем году.

11. На основе данных входного файла составить инвентарную ведомость склада, включив следующие данные: вид продукции, стоимость, сорт, количество. Вывести в новый файл информацию о той продукции, количество которой менее заданной величины.

12. На основе данных входного файла составить инвентарную ведомость склада, включив следующие данные: вид продукции, стоимость, сорт, количество. Вывести в новый файл инвентарную ведомость склада, увеличив стоимость каждого вида продукции на  $x\%$ .

13. На основе данных входного файла составить инвентарную ведомость игрушек, включив следующие данные: название игрушки, ее стоимость (в руб.), возрастные границы детей, для которых предназначена игрушка. Вывести в новый файл информацию о тех игрушках, которые предназначены для детей от  $N$  до  $M$  лет.

14. На основе данных входного файла составить инвентарную ведомость игрушек, включив следующие данные: название игрушки, ее стоимость (в руб.).

возрастные границы детей, для которых предназначена игрушка. Вывести в новый файл инвентарную ведомость игрушек, уменьшив стоимость каждого вида игрушек на  $x\%$ .

15. На основе данных входного файла составить список вкладчиков банка, включив следующие данные: ФИО, номер счета, сумма, год открытия счета. Вывести в новый файл информацию о тех вкладчиках, которые открыли вклад в текущем году.

16. На основе данных входного файла составить список вкладчиков банка, включив следующие данные: ФИО, номер счета, сумма, год открытия счета. Вывести в новый файл информацию о тех вкладчиках, сумма вклада которых превышает заданное значение.

17. На основе данных входного файла составить список студентов, включающий фамилию, факультет, курс, группу, пять оценок последней сессии. Вывести в новый файл информацию о тех студентах, которые сдали сессию на 4 и 5.

18. На основе данных входного файла составить список студентов, включающий фамилию, факультет, курс, группу, пять оценок. Вывести в новый файл информацию о тех студентах, которые имеют хотя бы одну двойку.

19. На основе данных входного файла составить список студентов, включающий ФИО, курс, группу, результат забега. Вывести в новый файл информацию о студентах заданной группы.

20. На основе данных входного файла составить список студентов, включающий ФИО, курс, группу, результат забега. Вывести в новый файл список студентов, удалив из него информацию о тех студентах, которые не выполнили норматив по бегу.

## Самостоятельная работа

**Задача 1.** Дано множество точек на плоскости. Выбрать из них такие четыре точки, которые составляют квадрат наибольшего периметра.

Исходные данные хранятся в файле *input.txt*, при этом координаты каждой точки записаны в отдельной строке через пробел. В файл *output.txt* вначале выводятся координаты найденных точек. Например:

```
input.txt
-1 1
1 1
3 1
1 3
1 4
1 -1
1 -2
```

```
output.txt
-1 1
1 3
3 1
1 -1
```

**Задача 2.** Определить радиус и центр окружности, проходящей через три различные точки заданного множества точек на плоскости и содержащей внутри себя наибольшее количество точек этого множества.

Исходные данные хранятся в файле *input.txt*, при этом координаты каждой точки записаны в отдельной строке через пробел. В файл *output.txt* вначале

выводятся координаты центра, а затем с новой строки радиус найденной окружности. Например:

*input.txt*

```
-3 0
-2 0
2 0
3 0
-1 1
1 1
0 3
3 3
0 -1
-1 -2
```

*output.txt*

```
0 0
3
```

# Глава 11

## СОРТИРОВКИ

Сортировкой или упорядочиванием списка объектов называется расположение этих объектов по возрастанию или убыванию согласно определенному линейному отношению порядка, такому, например, как отношение «≤» для целых чисел.

Будем далее предполагать, что сортируемые объекты являются записями, содержащими одно или несколько полей. Одно из полей, называемое ключом, имеет такой тип данных, что на нем определено отношение линейного порядка «≤». Например, это может быть целое или действительное число, строка. Задача сортировки состоит в упорядочивании записей таким образом, чтобы значения ключевого поля составляли неубывающую последовательность. Другими словами, записи  $r_1, r_2, \dots, r_n$  со значениями ключей  $k_1, k_2, \dots, k_n$  (не обязательно различными) надо расположить в порядке  $r_{i_1}, r_{i_2}, \dots, r_{i_n}$  таком, что  $k_{i_1} \leq k_{i_2} \leq \dots \leq k_{i_n}$ .

Существуют различные критерии оценки времени выполнения алгоритмов сортировки. Первой и наиболее общей мерой времени выполнения является количество шагов алгоритма, необходимых для упорядочивания  $n$  записей. Если размер записей большой, то перестановка записей занимает больше времени, чем все другие операции. Поэтому другой общей мерой служит количество перестановок записей, выполненных в ходе алгоритма.

В приведенных далее листингах программ будем использовать следующие обозначения:  $a$  — массив из  $n$  записей;  $key$  — одно из полей записей, которое является ключом;  $temp$  — переменная того же типа, что и элементы  $a$ . Каждый вид сортировки будет оформлен в виде отдельной функции.

### 11.1. Метод «пузырька»

Представим, что записи, подлежащие сортировке, хранятся в массиве, расположенном вертикально. Записи с малыми значениями ключевого поля более «легкие» и «всплывают» вверх наподобие пузырька. При первом проходе вдоль массива, начиная проход снизу, берется первая запись массива, и ее ключ поочередно сравнивается с ключами последующих записей. Если встречается запись с более «тяжелым» ключом, то эти записи меняются местами. При встрече с записью с более «легким» ключом эта запись становится эталоном для сравнения, и все

последующие записи сравниваются с этим новым, более «легким» ключом. В результате запись с наименьшим значением ключа оказывается в самом верху массива. Во время второго прохода вдоль массива находится запись со вторым по величине ключом, которая помещается под записью, найденной при первом проходе массива, т.е. на вторую сверху позицию, и т.д. Отметим, что во время второго и последующих проходов вдоль массива нет необходимости просматривать записи, найденные за предыдущие проходы, так как они имеют ключи, меньшие, чем у оставшихся записей. Другими словами, во время  $i$ -го прохода не проверяются записи, стоящие на позициях выше  $i$ . Заметим также, что после  $(n - 1)$ -го прохода последняя запись уже будет стоять на своем месте.

```
void sort(mas *a, int n)
{
    mas temp;
    int i, j;
    for(i = 0; i < n-1; i++)
        for (j = n-1; j>i; j--)
            if (a[j].key < a[j-1].key)
            {
                temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
}
```

### Замечание

Напомним, что в C++ нумерация элементов массива начинается с 0, поэтому в массиве из  $n$  элементов последний элемент будет с номером  $n - 1$ .

### Пример

Рассмотрим список студентов.

ФИО	Год рождения
Петров	1985
Эдуардов	1983
Кузнецова	1984
Антонов	1982
Семенова	1981
Власов	1983

Для этого примера объявление типов данных будет выглядеть следующим образом:

```
struct mas
{
    int key;      // ключевое поле
```

```

    char fio[30]; // информационное поле
};

mas *a; // указатель на нулевой элемент массива
// рабочая переменная, которая будет использоваться для
// перестановки местами двух элементов массива
mas temp;

```

Применим алгоритм «пузырька» для упорядочивания списка студентов по возрастанию года рождения. В табл. 11.1 показаны пять проходов алгоритма ( $n = 6$ ). Линии указывают позицию, выше которой записи уже упорядочены. После пятого прохода все записи, кроме последней, стоят на нужных местах. Но последняя запись не случайно оказалась последней: она также уже стоит на нужном месте. Поэтому сортировка закончена.

Таблица 11.1

Началь- ное положение	1-й про- ход	2-й про- ход	3-й про- ход	4-й про- ход	5-й про- ход
1985	1981	1981	1981	1981	1981
1983	1985	1982	1982	1982	1982
1984	1983	1985	1983	1983	1983
1982	1985	1983	1985	1983	1983
1981	1982	1984	1983	1985	1984
1983	1983	1983	1984	1984	1985

## 11.2. Сортировка вставками

Идея этого метода заключается в том, что на  $i$ -м этапе мы «вставляем» элемент  $a[i]$  в нужную позицию среди элементов  $a[1], a[2], \dots, a[i - 1]$ , которые уже упорядочены. До начала работы считаем упорядоченным (относительно самого себя) первый элемент. На первом проходе берем второй элемент и «вставляем» его относительно первого на нужное место: сравниваем ключи  $a[2]$  и  $a[1]$ , если ключ  $a[2]$  меньше ключа  $a[1]$ , то меняем записи местами. После этого считаем  $a[1]$  и  $a[2]$  упорядоченными (относительно друг друга). Не втором проходе берем третий элемент и «вставляем» его в нужную позицию среди элементов  $a[1], a[2], a[3]$ , сравнивая сначала его ключ с ключом  $a[2]$  (если ключ  $a[3]$  меньше ключа  $a[2]$ , то меняем местами), а потом (при необходимости) с ключом  $a[3]$ . И т.д., до последнего элемента. Так как сортировка началась со второго элемента, то всего будет  $n - 1$  проход.

Поскольку перемещение элементов происходит в цикле «по условию» (пока «нижний» меньше «верхнего»), то нужно следить, чтобы данный цикл корректно закончил свою работу в том случае, когда оче-

редной перемещаемый элемент будет меньше всех, которые стоят выше него (условие всегда будет истинным). Можно ввести элемент  $a[0]$ , чье значение ключа будет меньше значения ключа любого элемента  $a[1], a[2], \dots, a[n]$ . Обозначим ключевое значение элемента  $a[0]$  символом  $-\infty$ . Если такое значение нельзя применить, то при вставке  $a[i]$  в позицию  $j - 1$  нужно проверять, не будет ли  $j = 1$ . Если нет, тогда сравнивать элемент  $a[i]$ , который находится в позиции  $j$ , с элементом  $a[j - 1]$ .

```
void sort(mas *a, int n)
{
    mas temp;
    int i, j;
    for( i = 2; i <= n; i++)
    {
        j = i;
        while (a[j].key < a[j-1].key)
        {
            temp = a[j];
            a[j] = a[j-1];
            a[j-1] = temp;
            j--;
        }
    }
}
```

### Замечание

При заполнении массива в поле *key* нулевого элемента было записано значение  $-\text{MAXINT}$  (*<values.h>*), как того требовал алгоритм. А  $n$  записей были помещены в массив, начиная с первого элемента. Поэтому для выполнения алгоритма сортировки вставками потребуется одномерный массив размерностью  $n + 1$ .

### Пример

В табл. 11.2 показаны этапы алгоритма сортировки вставками, который используется для упорядочивания списка из предыдущего примера. После каждого этапа алгоритма элементы, расположенные выше линии, уже упорядочены, хотя между ними на последующих этапах могут быть вставлены элементы, которые на данном этапе расположены ниже линии.

Таблица 11.2

Начальное положение	1-й проход	2-й проход	3-й проход	4-й проход	5-й проход
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1985	1983	1983	1982	1981	1981
1983	1985	1984	1983	1982	1982
1984	1984	1985	1984	1983	1983

Окончание табл. 1.2

Начальное положение	1-й проход	2-й проход	3-й проход	4-й проход	5-й проход
1982	1982	1982	1985	1984	1983
1981	1981	1981	1981	1985	1984
1983	1983	1983	1983	1983	1985

### 11.3. Сортировка посредством выбора

На первом этапе среди всех элементов выполняется поиск записи с наименьшим ключом, которая меняется местами в первой, после чего считается упорядоченной. На втором этапе среди элементов, начиная со второго, ищется запись с наименьшим ключом и меняется местами со второй. На  $i$ -м этапе сортировки выбирается запись с наименьшим ключом среди записей  $a[1], \dots, a[n]$  и меняется местами с записью  $a[i]$ . В результате после  $i$ -го этапа все записи  $a[1], \dots, a[i]$  будут упорядочены. Всего таких проходов будет  $n - 1$ , так как после этого последняя запись также будет стоять на своем месте.

```
void sort(mas *a, int n)
{
    mas temp;
    int lowkey;
    int lowindex;
    int i, j;
    for (i = 0; i < n - 1; i++)
    {
        lowindex = i;
        lowkey = a[i].key;
        for (j = i + 1; j < n; j++)
            if (a[j].key < lowkey)
            {
                lowkey = a[j].key;
                lowindex = j;
            }
        temp = a[i];
        a[i] = a[lowindex];
        a[lowindex] = temp;
    }
}
```

#### Пример

В табл. 11.3 показаны этапы сортировки посредством выбора для списка из примера параграфа 11.1. Линия в таблице показывает, что элементы, расположенные выше нее, имеют наименьшие значения ключей и уже упорядочены.

Таблица 11.3

Начальное положение	1-й проход	2-й проход	3-й проход	4-й проход	5-й проход
1985	1981	1981	1981	1981	1981
	1983	1983	1982	1982	1982
	1984	1984	1984	1983	1983
	1982	1982	1983	1984	1983
	1981	1985	1985	1985	1984
	1983	1983	1983	1984	1985

Рассмотренные нами алгоритмы являются простыми схемами сортировки. Время работы этих алгоритмов пропорционально  $n^2$  как в среднем, так и в худшем случае. Если же сравнивать эти алгоритмы с точки зрения количества перестановок, то более предпочтительным оказывается алгоритм выбора. Количество перестановок в этом алгоритме пропорционально  $n$ , в то время как в первых двух —  $n^2$ .

Для больших  $n$  простые алгоритмы сортировки заведомо проигрывают алгоритмам со временем выполнения, пропорциональным  $n \log n$ . Значение  $n$ , начиная с которого быстрые алгоритмы становятся предпочтительнее, зависит от многих факторов. Быстрые алгоритмы более сложны в реализации и в данном пособии рассмотрены не будут.

Для небольших значений  $n$  рекомендуется применять простой в реализации алгоритм сортировки Шелла, который имеет временную сложность  $O(n^{1.5})$ . Этот алгоритм является обобщением алгоритма «пузырька».

## 11.4. Алгоритм сортировки Шелла

Массив  $a$  из  $n$  элементов упорядочивается следующим образом. На первом шаге упорядочиваются элементы  $n/2$  пар ( $a[i], a[n/2 + i]$ ) для  $1 \leq i \leq n/2$ ; на втором шаге упорядочиваются элементы в  $n/4$  группах из четырех элементов ( $a[i], a[n/4 + i], a[n/2 + i], a[3n/4 + i]$ ) для  $1 \leq i \leq n/4$ ; на третьем шаге упорядочиваются элементы в  $n/8$  группах из восьми элементов и т.д. На последнем шаге упорядочиваются элементы сразу во всем массиве  $a$ . На каждом шаге для упорядочивания элементов используется метод сортировки вставками.

```
void sort(mas *a, int n)
{
    mas temp;
    int i, j, incr = n / 2;
    while (incr > 0)
    {
        for (i = incr; i < n; i++)
            for (j = i; j >= incr && a[j] < a[j - incr]; j -= incr)
                a[j] = a[j - incr];
        incr /= 2;
    }
}
```

```

{
    j = i - incr;
    while (j >= 0)
        if (a[j].key > a[j + incr].key)
        {
            temp = a[j];
            a[j] = a[j + incr];
            a[j + incr] = temp;
            j = j - incr;
        }
        else j = -1;
    }
    incr = incr / 2;
}
}

```

## Пример

Рассмотрим этапы работы алгоритма сортировки Шелла для следующего примера: упорядочить по неубыванию следующие числа: 1, 7, 3, 2, 0, 5, 0, 8 (табл. 11.4).

На первом этапе рассматриваются следующие пары: 0-й и 4-й элементы со значением 1 и 0 соответственно, 1-й и 5-й элементы со значением 7 и 5, 2-й и 6-й элементы со значением 3 и 0, 3-й и 7-й элементы со значением 2 и 8. Всего четыре пары по два элемента ( $n = 8$ ). В каждой паре упорядочиваем элементы в порядке возрастания методом вставки. В данном случае, в первых трех парах элементы поменяются местами, в четвертой все останется по-прежнему. В результате получим пары (0, 1), (5, 7), (0, 3), (2, 8) (см. табл. 11.4). На втором этапе рассматриваются четверки: 0, 2, 4 и 6-й элементы со значениями 0, 0, 1, 3 соответственно, 1, 3, 5 и 7-й элементы со значениями 5, 2, 7, 8. После упорядочивания получим четверки: (0, 0, 1, 3), (2, 5, 7, 8). На последнем этапе рассматривается весь массив: (0, 2, 0, 5, 1, 7, 3, 8). После упорядочивания получаем (0, 0, 1, 2, 3, 5, 7, 8) — окончательный результат.

Таблица 11.4

Начальное положение	1 шаг	2 шаг	3 шаг
1	0	0	0
7	5	2	0
3	0	0	1
2	2	5	2
0	1	1	3
5	7	7	5
0	3	3	7
8	8	8	8

В листинге алгоритма Шелла использовалась убывающая последовательность шагов  $n/2, n/4, n/8, \dots, 2, 1$ . В общем случае алгоритм работает с любой убывающей последовательностью шагов, у которой последний шаг равен 1. Например, при  $n$ , кратном трем, можно использовать последовательность  $n/3, n/9, n/27, \dots, 3, 1$ .

## 11.5. Решение практических задач с использованием сортировок

1. В файле *input.txt* содержатся сведения о группе студентов в формате:

- номер группы;
- количество студентов в группе;
- запись о каждом студенте группы содержит следующие сведения: фамилия, имя, отчество, оценки по пяти предметам.

Переписать данные файла *input.txt* в файл *output.txt*, отсортировав их по убыванию средней оценки. Для каждого студента вычисленную среднюю оценку вывести в файл *output.txt*.

```
#include "fstream"
#include "string"
#include "iostream"
#include "iomanip"
using namespace std;
// открываем глобальные файловые потоки
ifstream in("input.txt");
ofstream out("output.txt");
struct mas
{
    string fam, name, secondname; // фамилия, имя, отчество
    int ses[5]; // оценки по пяти предметам
    double key; // средняя оценка
    void print();
};
void mas::print() // вывод данных в выходной поток
{
    out << setw(12) << left << fam << setw(10) << name
        << setw(15) << secondname;
    for (int i = 0; i < 5; i++)
        out << setw(3) << ses[i];
    out << setw(5) << key << endl;
}
// сортировка массива записей из n элементов методом «пузырька»
void sort(mas *a, int n)
{
    mas temp;
    int i, j;
    for (i = 0; i < n - 1; i++)
        for (j = n - 1; j > i; j--)
            if (a[j].key > a[j - 1].key) {
                temp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = temp;
            }
}
int main()
{
    int n = 0, m, i;
```

```

mas stud[20];
if (!in)
    cout << "Ошибка при открытии файла input.txt\n";
else
{
    in >> m; // считываем номер группы
    // считываем данные про всех студентов
    while (in.peek() != EOF)
    {
        in >> stud[n].fam;
        in >> stud[n].name;
        in >> stud[n].secondname;
        // считываем оценки и высчитываем средний балл
        stud[n].key = 0;
        for (i = 0; i < 5; i++)
        {
            in >> stud[n].ses[i];
            stud[n].key += stud[n].ses[i];
        }
        stud[n].key /= 5;
        n++;
    }
    sort(stud, n); // сортируем массив записей
    // выводим отсортированные данные в файл output.txt
    out << m << endl;
    for (i = 0; i < n; i++)
        stud[i].print();
}
in.close();
out.close(); // закрываем файлы
return 0;
}

```

Результат работы программы:

#### ***input.txt***

```

111
Иванов Иван Иванович 5 4 5 4 5
Иванцов Сергей Петрович 5 4 4 5 4
Иванова Нина Юрьевна 5 5 5 5 5
Смирнова Анна Дмитриевна 3 4 3 3 3
Сидоров Андрей Григорьевич 4 3 4 3 4

```

#### ***output.txt***

```

111
Иванова   Нина   Юрьевна      5  5  5  5  5
Иванов    Иван   Иванович     5  4  5  4  5
4.6
Иванцов    Сергей  Петрович    5  4  4  5  4
4.4
Сидоров    Андрей  Григорьевич 4  3  4  3  4
3.6
Смирнова   Анна   Дмитриевна 3  4  3  3  3
3.2

```

**2.** Данна матрица размерностью  $n \times n$ , содержащая целые числа. Отсортировать каждую строчку матрицы по возрастанию элементов, используя алгоритм выбора.

```
#include "fstream"
#include "iostream"
#include "iomanip"
using namespace std;
ifstream in("input.txt");
ofstream out("output.txt");
// сортировка одномерного массива методом выбора
void sort(int *a, int n)
{
    int temp;
    int lowindex, lowkey, i, j;
    for (i = 0; i < n - 1; i++)
    {
        lowindex = i;
        lowkey = a[i];
        for (j = i + 1; j < n; j++)
            if (a[j] < lowkey)
            {
                lowkey = a[j];
                lowindex = j;
            }
        temp = a[i];
        a[i] = a[lowindex];
        a[lowindex] = temp;
    }
}
int main()
{
    int n, m, i, j;
    int a[10][10];
    // ввод данных из файла input.txt
    in >> n >> m;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            in >> a[i][j];
    // сортируем каждую строку двумерного массива с помощью сортировки
    // методом выбора
    for (i = 0; i < n; i++)
        sort(a[i], m);
    // выводим обработанные данные в файл output.txt
    out << n << '\t' << m << '\n';
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
            out << setw(5) << a[i][j];
        out << '\n';
    }
    in.close();
    out.close(); // закрываем файлы
    return 0;
}
```

Результат работы программы:

*input.txt*

```
4 5
23 54 65 -9 0
8 96 -4 -7 6
100 12 90 1 2
2 -1 4 -5 -12
```

*output.txt*

```
4 5
-9 0 23 54 65
-7 -4 6 8 96
1 2 12 90 100
-12 -5 -1 2 4
```

3. Данна матрица размерностью  $n \times n$ , содержащая целые числа. Отсортировать каждый столбец матрицы по убыванию элементов методом вставки.

```
#include "fstream"
#include "iostream"
#include "iomanip"
using namespace std;
ifstream in("input.txt");
ofstream out("output.txt");
// сортировка одномерного массива методом вставки (по убыванию)
void sort(int *a, int n)
{
    int temp;
    int i, j;
    for (i = 2; i <= n; i++)
    {
        j = i;
        while (a[j] < a[j - 1])
        {
            temp = a[j];
            a[j] = a[j - 1];
            a[j - 1] = temp;
            j--;
        }
    }
}
int main()
{
    int n, m, i, j;
    int a[10][10];
    // Ввод данных из файла
    in >> n >> m;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            in >> a[i][j];
    int b[10];
    // каждый столбец массива a копируем в массив b, сортируем
    // массив b по убыванию элементов методом выбора, затем
    // копируем элементы массива b в обрабатываемый столбец массива a
```

```

for (j = 0; j < m; j++)
{
    for (i = 0; i < n; i++)
        b[i] = a[i][j];
    sort(b, n);
    for (i = 0; i < n; i++)
        a[i][j] = b[i];
}
// выводим обработанные данные в файл output.txt
out << n << '\t' << m << '\n';
for (i = 0; i < n; i++)
{
    for (j = 0; j < m; j++)
        out << setw(5) << a[i][j];
    out << '\n';
}
in.close();
out.close();
return 0;
}

```

Результат работы программы:

**input.txt**

```

4 5
23 54 65 -9 0
8 96 -4 -7 6
100 12 90 1 2
2 -1 4 -5 -12

```

**output.txt**

```

4 5
100 96 90 1 6
23 54 65 -5 2
8 12 4 -7 0
2 -1 -4 -9 -12

```

## Упражнения

- В файле *input.txt* содержатся сведения о группе студентов в формате:  
 — номер группы;  
 — запись о каждом студенте группы содержит следующие сведения: фамилия, имя, отчество, год рождения, оценки по пяти предметам.  
 Переписать данные файла *input.txt* в файл *output.txt*, отсортировав их:
  - по убыванию средней оценки методом вставки (среднюю оценку вывести в файл *output.txt* для каждого студента);
  - по возрастанию средней оценки методом выбора (среднюю оценку вывести в файл *output.txt* для каждого студента);
  - по убыванию средней оценки алгоритмом Шелла (среднюю оценку вывести в файл *output.txt* для каждого студента);
  - по убыванию суммы оценок методом «пузырька» (сумму оценок вывести в файл *output.txt* для каждого студента);

- 5) по убыванию суммы оценок методом вставки (сумму оценок вывести в файл *output.txt* для каждого студента);
- 6) по возрастанию суммы оценок методом выбора (сумму оценок вывести в файл *output.txt* для каждого студента);
- 7) по убыванию суммы оценок алгоритмом Шелла (сумму оценок вывести в файл *output.txt* для каждого студента);
- 8) в алфавитном порядке по фамилии методом «пузырька»;
- 9) в алфавитном порядке по фамилии методом вставки;
- 10) в алфавитном порядке по фамилии методом выбора;
- 11) в алфавитном порядке по фамилии алгоритмом Шелла;
- 12) в алфавитном порядке по фамилии, а затем по возрастанию года рождения методом «пузырька»;
- 13) в алфавитном порядке по фамилии, а затем по убыванию года рождения методом вставки;
- 14) в алфавитном порядке по фамилии, а затем по возрастанию года рождения методом выбора;
- 15) в алфавитном порядке по фамилии, а затем по убыванию года рождения алгоритмом Шелла;
- 16) в алфавитном порядке по фамилии, имени, отчеству методом «пузырька»;
- 17) в алфавитном порядке по фамилии, имени, отчеству методом вставки;
- 18) в алфавитном порядке по фамилии, имени, отчеству методом выбора;
- 19) в алфавитном порядке по фамилии, имени, отчеству алгоритмом Шелла;
- 20) в алфавитном порядке по фамилии, имени, отчеству, а затем по убыванию года рождения алгоритмом Шелла;

2. Данна матрица размерностью  $n \times n$ , содержащая целые числа. Отсортировать:

- 1) каждую строчку матрицы по убыванию элементов методом «пузырька»;
- 2) каждую строчку матрицы по убыванию элементов алгоритмом Шелла;
- 3) каждую строчку матрицы по убыванию элементов методом вставки;
- 4) каждый столбец матрицы по возрастанию элементов методом выбора;
- 5) каждый столбец матрицы по возрастанию элементов алгоритмом Шелла;
- 6) каждый столбец матрицы по возрастанию элементов методом «пузырька»;
- 7) диагонали матрицы, параллельные главной, по убыванию элементов методом вставки;
- 8) диагонали матрицы, параллельные главной, по убыванию элементов методом выбора;
- 9) диагонали матрицы, параллельные главной, по убыванию элементов алгоритмом Шелла;
- 10) диагонали матрицы, параллельные главной, по убыванию элементов методом «пузырька»;
- 11) диагонали матрицы, параллельные побочной, по возрастанию элементов методом выбора;
- 12) диагонали матрицы, параллельные побочной, по возрастанию элементов алгоритмом Шелла;
- 13) диагонали матрицы, параллельные побочной, по возрастанию элементов методом «пузырька»;
- 14) диагонали матрицы, параллельные побочной, по возрастанию элементов методом вставки;
- 15) каждый столбец матрицы с номером  $2i$  по убыванию элементов, а с номером  $2i+1$  — по возрастанию элементов методом «пузырька»;
- 16) каждый столбец матрицы с номером  $2i$  по возрастанию элементов, а с номером  $2i+1$  — по убыванию элементов методом вставки;

17) диагонали матрицы, расположенные выше главной, по убыванию элементов, а диагонали матрицы, расположенные ниже главной, по возрастанию элементов методом выбора;

18) диагонали матрицы, расположенные выше главной, по возрастанию элементов, а диагонали матрицы, расположенные ниже главной, по убыванию элементов алгоритмом Шелла;

19) диагонали матрицы, расположенные выше побочной, по убыванию элементов, а диагонали матрицы, расположенные ниже побочной, по возрастанию элементов методом выбора;

20) диагонали матрицы, расположенные выше побочной, по возрастанию элементов, а диагонали матрицы, расположенные ниже побочной, по убыванию элементов методом вставки.

## Самостоятельная работа

1. Данна последовательность, состоящая из  $N$  целых чисел. Отсортировать ее, используя алгоритм:

- 1) быстрой сортировки;
- 2) пирамидальной сортировки;
- 3) «карманной сортировки»;
- 4) поразрядной сортировки;
- 5) сортировки подсчетом;
- 6) сортировки слиянием.

2. Предположим, что необходимо отсортировать список элементов, состоящий из уже упорядоченной последовательности элементов, которые следует за некоторыми «случайными» элементами. Какой из рассмотренных в этой главе методов сортировки или изученных вами самостоятельно методов наиболее подходит для этого решения?

3. Алгоритм называется устойчивым, если он сохраняет исходный порядок следования элементов с одинаковыми элементами ключей. Какие из рассмотренных в этой главе методов сортировки или изученных вами самостоятельно методов являются устойчивыми?

# Глава 12

## КЛАСС-КОНТЕЙНЕР ВЕКТОР

В стандарт C++ входит библиотека стандартных шаблонов STL (*Standard Template Library*), которая является достаточно мощным и удобным инструментом для хранения и обработки данных. Основными элементами библиотеки STL являются контейнеры, итераторы и алгоритмы.

Контейнер — это объект, предназначенный для хранения других объектов. Все объекты в контейнере имеют один тип. Контейнер является шаблоном класса, а шаблоны позволяют создать одно определение класса или функции, которое впоследствии можно применить для ряда типов. Таким образом, можно создать контейнер, содержащий строки, целые числа или объекты какого-то другого типа. Непосредственно для контейнеров определено небольшое количество операций, однако для работы с ними можно использовать множество дополнительных операций, реализованных в библиотеке алгоритмов STL.

В библиотеке стандартных шаблонов определены два вида контейнеров — последовательные и ассоциативные. В последовательных контейнерах элементы хранятся в порядке их поступления, и доступ к ним осуществляется по номеру позиции. В данной главе пособия будет рассмотрен класс последовательного контейнера *vector*. В ассоциативных контейнерах доступ к элементам осуществляется с помощью ключей.

Итератор — это тип, позволяющий обращаться к хранимым в контейнере элементам, перемещаясь от одного к другому. Можно сказать, что итераторы по отношению к контейнерам играют роль индексов или указателей. В библиотеке тип итератора определен для каждого стандартного контейнера, а вот индексирование поддерживают не все контейнеры.

Алгоритмы, как уже говорилось выше, выполняют операции над элементами контейнеров.

### 12.1. Работа с векторами

Вектор — это тип данных, похожий на массив, служащий тем же целям, что и массив, но имеющий существенные отличия. Подобно массиву, вектор имеет базовый тип и содержит набор значений этого типа. Однако массив имеет фиксированную размерность (длину), которая

не может ни увеличиваться, ни уменьшаться, а длина вектора может динамически изменяться. Кроме того, описание вектора и работы с ним отличаются от описания и работы с массивом.

#### Замечание

Для того чтобы использовать в программе вектор, к ней необходимо подключить заголовочный файл "vector".

Чтобы объявить объект типа *vector*, необходимо указать его имя и базовый тип элементов:

```
vector <базовый_тип_элементов> имя_объекта;
```

#### Замечание

В данном описании объекта типа *vector* скобки <> являются обязательными элементами.

Например, объявить вектор, состоящий из целых чисел, можно следующим образом:

```
vector <int> iVec;
```

#### Замечание

Ключевое слово *vector* определяет не имя типа, а имя шаблона, который можно использовать для определения вектора, способного хранить наборы любых типов. Следовательно, тип объекта *iVec* определяет выражение *vector <int>*. Более того, выражение *vector <int>* является не просто именем типа данных, это — имя класса. Аналогичные классы существуют и для других базовых типов данных. Таким образом, приведенное выше объявление создает объект *iVec* класса *vector <int>*. При объявлении объекта *iVec* используется конструктор по умолчанию, который создает пустой вектор, т.е. вектор с нулевой длиной.

Точно так же, как и в массиве, элементы вектора нумеруются с 0. Для обращения к ним можно записать имя вектора и индекс в квадратных скобках, например *iVec[1]*. Однако это выражение не может использоваться для инициализации вектора (т.е. для присвоения начального значения). Оно применяется только для изменения значения уже существующего элемента или для какого-либо другого использования этого значения, например для вывода.

Для инициализации вектора можно использовать конструктор или последовательно добавлять в вектор элементы с помощью функции-члена *push\_back*. Рассмотрим эти способы подробнее.

В классе *vector* определено несколько конструкторов, которые можно использовать при определении и инициализации объектов вектора:

```
// пустой вектор v1, который будет содержать объекты типа t
vector <t> v1;
// вектор v1 - копия вектора v2
vector <t> v1(v2);
// вектор v1 содержит n элементов типа t со значением i
vector <t> v1(n, i);
// вектор v1 содержит n элементов типа t, значение
// которым присваивается в зависимости от типа t
vector <t> v1(n);
```

### Замечание

Если элементы вектора имеют базовый тип, например `int`, то при использовании последнего конструктора элементы вектора примут значение 0. Если элементы являются объектами класса и для них определены собственные конструкторы, то для инициализации используется стандартный конструктор класса.

Главным преимуществом вектора является возможность динамически изменять размер по мере добавления в него элементов. Хотя количество элементов вектора можно задать заранее, обычно удобнее создавать пустой вектор и добавлять в него элементы по мере надобности. По мере добавления элементов размер вектора будет увеличиваться.

Элементы добавляются в вектор с помощью члена-функции `push_back` в определенном порядке: сначала в позицию 0, затем в позицию 1, позицию 2 и т. д. Таким образом, добавление элемента производится фактически в конец вектора.

Например, добавить элемент со значением  $i$  в конец вектора `iVec` можно следующим образом:

```
iVec.push_back(i);
```

Чтобы добавить в вектор элементы со значениями 1, 2, 3, 4, 5, можно использовать следующий цикл:

```
for (int i = 1; i <= 5; i++)
    iVec.push_back(i);
```

Вывести элементы полученного вектора на экран можно, используя обращение по индексу, как в массиве:

```
for (unsigned int i = 0; i < 5; i++)
    cout << iVec[i] << endl;
```

Количество элементов вектора называется его *размером*. Чтобы узнать размер вектора, можно воспользоваться функцией-членом `size()`. Тогда вывод на экран всех элементов вектора `iVec` можно выполнить следующим образом:

```
for (unsigned int i = 0; i < iVec.size(); i++)
    cout << iVec[i] << endl;
```

Следует отметить, что функция `size` возвращает значение типа `unsigned int`, и тип индекса, с помощью которого мы обращаемся к элементам вектора, тоже `unsigned int`. Поэтому переменная `i` объявлена типом `unsigned int`, а не `int`.

#### Замечание

Если вы объявили переменную `i` типом `int`, то компилятор выдаст предупреждение о том, что будет выполняться автоматическое преобразование типов.

Для вектора, точно так же как и для строк, можно использовать тип `size_type`. Поэтому вывод элементов вектора на экран можно организовать с помощью следующего цикла:

```
for (vector<int>::size_type i = 0; i < iVec.size(); i++)
    cout << iVec[i] << endl;
```

Кроме размера, для вектора существует еще такое понятие, как емкость. В любой момент времени вектор имеет определенную емкость, т.е. определено количество элементов, для которых на этот момент выделена память. Емкость вектора можно узнать с помощью функции-члена `capacity()`.

#### Замечание

Не путайте емкость вектора и его размер! Размер определяет количество элементов в заполненной части вектора, а емкость — это количество элементов, для которых выделена память. Емкость обычно больше размера, а не наоборот.

Когда емкость вектора исчерпана, она автоматически увеличивается. Насколько — зависит от конкретной реализации, но всегда на большую величину, чем нужно в настоящий момент (как правило, она удваивается). Поскольку увеличение емкости вектора является достаточно сложной задачей, выделение памяти большими блоками эффективнее, чем маленькими.

Проиллюстрируем вышесказанное примером.

```
#include "vector"
#include "iostream"
using namespace std;

int main()
{
    vector<int> iVec;
    cout << "Vector empty" << endl;
    cout << "size:" << iVec.size() << endl;
    cout << "capacity:" << iVec.capacity() << endl;
    for (int i = 1; i <= 5; i++)
        iVec.push_back(i);
```

```
cout << "Vector is (1,2,3,4,5)" << endl;
cout << "size:" << iVec.size() << endl;
cout << "capacity:" << iVec.capacity() << endl;
for (int i = 6; i <= 10; i++)
    iVec.push_back(i);
cout << "Vector is (1,2,3,4,5,6,7,8,9,10)" << endl;
cout << "size:" << iVec.size() << endl;
cout << "capacity:" << iVec.capacity() << endl;
return 0;
}
```

После выполнения данной программы на экране будет выведено:

```
Vector empty
size: 0
capacity: 0
Vector is (1,2,3,4,5)
size: 5
capacity: 6
Vector is (1,2,3,4,5,6,7,8,9,10)
size: 10
capacity: 13
```

Проверить, является ли вектор пустым, можно с помощью член-функции *empty()*. Она возвращает значение *true*, если вектор пуст, и *false* в противном случае.

Сравнивать векторы можно с помощью операций `==`, `!=`, `<`, `>`, `<=`, `>=`. Эти операции имеют обычное назначение. Так, например, операция `==` возвращает значение *true*, если операнды-вектора равны (имеют одинаковое количество одинаковых элементов, которые идут в одинаковом порядке), и *false* в противном случае.

#### Замечание

Самостоятельно проверьте работу остальных операций.

Вставлять элементы в вектор можно с помощью функции *insert*, а удалять — с помощью *erase*. Рассмотрим их подробнее.

#### Замечание

Отметим, что не следует злоупотреблять данными функциями, так как при каждой вставке или удалении будет изменяться структура вектора — он будет сжиматься или разжиматься, что будет снижать эффективность выполнения программы.

Функция *insert(iter, value)* вставляет элемент со значением *value* в позицию, которую указывает итератор *iter*. Так, например, для заданного вектора *iVec*, элементами которого являются целые числа, вставить элемент 10 во вторую позицию можно следующим образом:

```
iVec.insert(iVec.begin() + 2, 10);
```

## Замечание

Функция `begin()` возвращает итератор, которой позволяет обратиться к первому элементу вектора. Более подробно итераторы будут рассмотрены в следующем параграфе.

Функция `erase(iter)` удаляет элемент из позиции, на которую указывает итератор `iter`. Например, удалить второй элемент из вектора можно следующим образом:

```
iVec.erase(iVec.begin() + 2);
```

Метод `erase(iter1, iter2)` удаляет элементы из диапазона `iter1, iter2`, где `iter1, iter2` — итераторы.

## 12.2. Итераторы

Кроме индексирования, для доступа к элементам вектора библиотека предоставляет еще один способ — **итератор**. Свой собственный итератор определен в каждом из классов-контейнеров, в том числе и в классе `vector`. Для нашего примера (вектора из целых чисел) итератор будет определяться следующим образом:

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

В данном примере мы определили переменную `iter` типа `iterator` для класса `vector <int>`.

## Замечание

Напомним, что тип с именем `iterator` определен в каждом библиотечном классе-контейнере.

В классе каждого контейнера определены две функции, `begin()` и `end()`, которые возвращают итератор. Итератор, возвращаемый функцией `begin()`, позволяет обратиться к первому элементу контейнера, если он есть. Обращение `iVec.begin()`, если вектор не пуст, эквивалентно обращению `iVec[0]`.

Итератор, возвращаемый функцией `end()`, указывает на элемент, следующий за последним в векторе. Иногда говорят, что он указывает на конец вектора, но на самом деле, если им воспользоваться, произойдет выход за границу вектора. Этот итератор используется как граница при переборе элементов вектора. Если вектор пуст, функции `begin()` и `end()` возвращают одинаковый итератор.

**Операции с итераторами.** Операции с переменными типа `iterator` позволяют получить доступ к элементу, на который указывает итератор, а также переместить итератор с одного элемента на другой.

Для доступа к элементу, на который указывает итератор, используется операция обращения к значению `*`, т.е. `*iter` возвращает элемент, на который указывает итератор `iter`.

Чтобы переместить итератор на следующий элемент в контейнере, используется оператор инкремента `++`, т.е. если `iter` указывал на первый элемент вектора, то после выполнения `++iter` он будет указывать на второй элемент. Аналогично с итераторами используется оператор декремента `--`.

Итераторы векторов поддерживают также другие арифметические операции. Например, к итератору можно прибавить (из него вычесть) целочисленное значение:

```
iter + n;
```

или

```
iter - n;
```

В результате получается новый итератор, который указывает на  $n$  элементов вперед (или назад) от исходного значения итератора `iter`. Результат сложения должен указывать на элемент вектора или одну из его границ. Типом полученного значения является обычно `size_type` или `difference_type`. Последний также определен в классе вектора и является знаковым.

Кроме того, можно сложить или вычесть два итератора, относящиеся к одному вектору. Результатом будет знаковое целочисленное значение типа `difference_type`. Проиллюстрируем вышесказанное примером вывода элементов вектора `iVec`:

```
for (vector <int>::iterator iter = iVec.begin();  
     iter <iVec.end(); iter++)  
    cout << *iter << endl;
```

#### Замечание

Этот фрагмент безошибочно сработает и с пустым вектором. В этом случае итератор, который возвращает функция `begin()`, совпадает с итератором, который возвращает функция `end()`, и после первой проверки условия цикл выполнятся не будет.

## 12.3. Алгоритмы STL

В библиотечных контейнерах определен некоторый набор функций, предназначенный для их обработки. В дополнение к нему STL предоставляет набор алгоритмов, которые не зависят от конкретного типа контейнера и выполняют такие действия, как поиск, замена, удаление элементов в контейнерах, сортировка элементов контейнеров и многие другие. Аргументами этих алгоритмов служат итераторы. Для обеспечения доступа к алгоритмам STL нужно подключить заголовочный файл "`algorithm`".

## Замечание

Алгоритмы STL можно применять и к массивам, если в качестве итераторов использовать указатели на элементы массива.

Рассмотрим наиболее интересные алгоритмы для работы с векторами. Если не будет оговорено другого, то в качестве параметров алгоритма мы будем использовать итераторы *first* и *last*, обозначающие соответственно начало и конец диапазона, в котором алгоритмом производятся соответствующие действия.

Алгоритм *find(first, last, value)* ищет в заданном диапазоне первый элемент, значение которого равно *value*. Возвращает итератор на найденный элемент.

Алгоритм *count(first, last, value)* подсчитывает, сколько раз в заданном диапазоне встречается элемент со значением *value*. Возвращает найденное количество.

Алгоритм *sort(first, last)* сортирует объекты в заданном диапазоне.

Алгоритм *replace(first, last, old, new)* в заданном диапазоне заменяет все объекты, равные *old*, объектами, равными *new*.

Алгоритм *remove(first, last, value)* удаляет из заданного диапазона все объекты, равные *value*. Реализуется это следующим образом. Элементы, которые надо удалить, перемещаются в конец контейнера, и алгоритм возвращает итератор, который указывает на начало последовательности удаляемых элементов.

Алгоритм *min\_element(first, last, value)* возвращает итератор, указывающий на наименьший элемент в заданном диапазоне.

Алгоритм *max\_element(first, last, value)* возвращает итератор, указывающий на наибольший элемент в заданном диапазоне.

Алгоритм *iter\_swap(iter1, iter2)* меняет местами объект, на который указывает итератор *iter1*, с объектом, на который указывает итератор *iter2*.

Рассмотрим следующие примеры.

## Пример 1

Дана последовательность из *n* элементов. Поменять местами максимальный и минимальный элемент.

```
#include "iostream"
#include "algorithm"
#include "vector"
using namespace std;
int main()
{
    vector<int> iVec;
    int x, n;
    cout << "n = ";
    cin >> n;
```

```

// В цикле формируем вектор из n элементов, значения которых
// вводятся с клавиатуры. Добавление элементов производится
// в конец вектора с помощью функции push_back
for (int i = 0; i < n; i++)
{
    cout << "Введите элемент с номером " << i << endl;
    cin >> x;
    iVec.push_back(x);
}
// с помощью алгоритма min_element находим итератор iterMin,
// который указывает на минимальный элемент в векторе
vector<int>::iterator iterMin =
    min_element(iVec.begin(), iVec.end());
// выводим минимальный элемент на экран
cout << "min = " << *iterMin << endl;
// с помощью алгоритма max_element находим итератор iterMax,
// который указывает на максимальный элемент
vector<int>::iterator iterMax =
    max_element(iVec.begin(), iVec.end());
// выводим максимальный элемент
cout << "max = " << *iterMax << endl;
// с помощью алгоритма iter_swap(iterMin, iterMax) меняем
// местами элемент, на который указывает iterMin, и элемент,
// на который указывает iterMax
iter_swap(iterMin, iterMax);
// выводим на экран измененный вектор
for (vector<int>::iterator iter = iVec.begin(); iter < iVec.
end(); iter++)
    cout << *iter << endl;
return 0;
}

```

---

## Пример 2

Дана последовательность из n элементов. Удалить все элементы, равные 0.

```

#include "iostream"
#include "algorithm"
#include "vector"
using namespace std;
int main()
{
    vector<int> iVec;
    int x, n;
    cout << "n = ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cout << "vvedite " << i << " el vectora" << endl;
        cin >> x;
        iVec.push_back(x);
    }
    // элементы, равные нулю, перемещаются в конец вектора
    // и располагаются, начиная с позиции, на которую
    // указывает iEnd

```

```

vector<int>::iterator iEnd =
    remove(iVec.begin(), iVec.end(), 0);
// тогда вывести нам нужно элементы
// с начала вектора до позиции iEnd
for (vector<int>::iterator iter = iVec.begin();
     iter < iEnd; iter++)
    cout << *iter << endl;
return 0;
}

```

Чтобы физически удалить нулевые элементы из вектора, можно после функции *remove* использовать функцию *erase*. Тогда конец программы будет выглядеть следующим образом:

```

vector <int>::iterator iEnd =
    remove(iVec.begin(), iVec.end(), 0);
iVec.erase(iEnd,iVec.end());
for (vector <int>::iterator iter = iVec.begin();
     iter<iVec.end(); iter++)
    cout << *iter << endl;

```

---

**Добавление *\_if* к названию алгоритма.** Некоторые алгоритмы имеют версии с окончанием *\_if* (например, *find\_if*, *count\_if*, *replace\_if*, *remove\_if*). Для версий функций с окончанием *\_if* требуется дополнительный параметр, который называется *предикатом* и который в свою очередь также является функцией.

Например, чтобы найти в контейнере *iVec* элемент со значением 5, мы будем использовать следующий вызов функции *find*:

```
vector <int>::iterator iter = find(iVec.begin(), iVec.end(), 5);
```

Рассмотрим теперь, как найти в контейнере *iVec* первый положительный элемент. Будем использовать для этого функцию *find\_if*, которая в качестве одного из аргументов использует предикат. Предикат — это функция, которая возвращает значение *true*, если условие выполняется и *false* в противном случае. В нашем случае условием будет «аргумент больше нуля». Тогда функцию можно задать следующим образом:

```

bool Pred(int x)
{
    return (x < 0);
}

```

Обращение к *find\_if* для решения нашей задачи можно записать следующим образом:

```
vector <int>::iterator it = find_if(iVec.begin(), iVec.end(), Pred);
```

### Пример 3

В заданном наборе целых чисел все отрицательные элементы заменить на 0, используя алгоритм *replace\_if*.

### *Вариант 1. Работа с массивом.*

*Замечание.* Если мы описали одномерный массив  $a$  из  $n$  элементов, то обратиться к его элементам можно через индекс или через указатель. Например, чтобы обратиться к нулевому элементу массива, можно написать  $a[0]$  или  $*a$ . Чтобы обратиться к элементу с номером  $i$  —  $a[i]$  или  $*(a + i)$  соответственно. Чтобы обратиться к последнему элементу, нужно написать  $a[n - 1]$  или  $*(a + n - 1)$ . Таким образом, указатель на последний элемент в массиве будет задаваться выражением  $a + n - 1$ .

```
#include "iostream"
#include "algorithm"
using namespace std;
// функция возвращает значение true,
// если аргумент отрицательный,
// в противном случае возвращает значение false
bool isNeg(int x)
{
    return (x < 0);
}
int main()
{
    int n;
    int a[10];
    cout << "n = ";
    cin >> n for (int i = 0; i < n; i++) // вводим элементы массива
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }
/*
    Все элементы в диапазоне от a (указатель на первый элемент
    массива) до a + n - 1 (указатель на последний элемент массива),
    для которых выполняется предикат isNeg (т.е. элементы являются
    отрицательными), заменяем 0
*/
    replace_if(a, a + n - 1, isNeg, 0);
    for (int i = 0; i < n; ++i) // выводим измененный массив
        cout << "a[" << i << "] = " << a[i] << endl;
    return 0;
}
```

### *Вариант 2. Работа с вектором.*

*Замечание.* В данном случае при обращении к алгоритму *replace\_if* первым и вторым аргументами будут итераторы, указывающие на первый и последний элемент вектора соответственно.

```
#include "iostream"
#include "algorithm"
#include "vector"
using namespace std;
bool isNeg(int x)
{
    return (x < 0);
}
```

```

int main()
{
    vector<int> iVec;
    int x, n;
    cout << "n = ";
    cin >> n;
    // формируем вектор из n элементов,
    // значения которых вводятся с клавитуры
    for (int i = 0; i < n; i++)
    {
        cout << "Введите элемент вектора с номером " << i << endl;
        cin >> x;
        iVec.push_back(x);
    }
/*
Все элементы вектора в диапазоне от iVec.begin() – итератор,
указывающий на первый элемент вектора, до iVec.end() – итератор,
указывающий на конец вектора, для которых выполняется предикат
isNeg (т.е. элементы вектора являются отрицательными), заменяем 0
*/
    replace_if(iVec.begin(), iVec.end(), isNeg, 0);
    // выводим измененный вектор
    for (vector<int>::iterator iter = iVec.begin(); iter < iVec.
end(); iter++)
        cout << *iter << endl;
    return 0;
}

```

---

**Обобщенные числовые алгоритмы.** В STL существуют так называемые *обобщенные числовые алгоритмы*, для работы с которыми необходимо подключить заголовочный файл "numeric". Рассмотрим для примера один из них.

Алгоритм *accumulate(first, last, init)* считает сумму элементов из заданного диапазона, где *init* — это начальное значение этой суммы. Алгоритм возвращает результат суммирования. Начальное значение указывает функции тип суммируемых элементов и оно должно совпадать с типом элементов контейнера или допускать преобразование в тип элементов контейнера. Тогда подсчитать сумму элементов вектора *iVec*, состоящего из целых чисел, можно следующим образом:

```
int sum = accumulate(iVec.begin(), iVec.end(), 0);
```

## Упражнения

- I. Дано последовательность целых чисел.
  1. Заменить все положительные элементы на *x*.
  2. Заменить все четные элементы на *x*.
  3. Заменить все элементы, попадающие в интервал  $[a, b]$ , нулем.
  4. Заменить все элементы, значения которых равны *x* на *y*.
  5. Заменить все двухзначные числа на *x*.

6. Заменить все простые числа на  $x$ .
7. Подсчитать количество четных элементов.
8. Подсчитать количество элементов, кратных девяти.
9. Подсчитать количество элементов, не попадающих в заданный интервал.
10. Определить, является ли сумма элементов двухзначным числом.
11. Определить, является ли сумма элементов простым числом.
12. Подсчитать количество максимальных элементов.
13. Заменить все максимальные элементы нулями.
14. Заменить все минимальные элементы данным числом  $x$ .
15. Поменять местами максимальный и первый элементы.
16. Подсчитать сумму элементов, расположенных между максимальным и минимальным элементами (минимальный и максимальный элементы в массиве единственные). Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
17. Подсчитать сумму элементов, расположенных между минимальным и максимальным элементами (минимальный и максимальный элементы в массиве единственные). Если минимальный элемент встречается позже максимального, то выдать сообщение об этом.
18. Поменять местами первый минимальный и последний элементы.
19. Выполнить перестановку элементов по правилу: первый с последним, второй с предпоследним и т.д.
20. Выполнить перестановку элементов по правилу: первый со вторым, третий с четвертым и т.д.

## II. Данна последовательность целых чисел.

1. Удалить из массива все четные числа.
2. Удалить из массива все максимальные элементы.
3. Удалить из массива все числа, значения которых попадают в данный интервал.
4. Удалить из массива все элементы, последняя цифра которых равна  $x$ .
5. Удалить из массива элементы с номера  $k_1$  по номер  $k_2$ .
6. Удалить из массива каждый  $k$ -й по счету элемент.
7. Удалить из массива последний минимальный элемент.
8. Вставить новый элемент перед первым отрицательным элементом.
9. Вставить новый элемент после последнего положительного.
10. Вставить новый элемент перед всеми четными элементами.
11. Вставить новый элемент после всех элементов, которые заканчиваются на заданную цифру.
12. Вставить новый элемент после всех элементов, кратных своему номеру.
13. Перед каждым  $k$ -м по счету элементом вставить 0.
14. Удвоить каждый  $k$ -й элемент.
15. Удвоить каждый отрицательный элемент.
16. После каждого  $k$ -го по счету элемента вставить новое значение.
17. Удалить из массива все элементы, в записи которых все цифры различны.
18. Удалить из массива повторяющиеся элементы, оставив только их первые вхождения.
19. Вставить новый элемент после всех максимальных.
20. Вставить новый элемент перед всеми элементами, в записи которых есть данная цифра.

# Глава 13

## ИСКЛЮЧЕНИЯ

Язык C++, как и многие другие объектно-ориентированные языки, реагирует на ошибки и ненормальные ситуации с помощью механизма обработки исключений. **Исключение** — это объект, генерирующий информацию о «необычном программном происшествии». При этом механизм обработки исключений в C++ не поддерживает обработку асинхронных событий, таких как ошибки оборудования, или обработку прерываний, например нажатие комбинаций клавиш на клавиатуре. Механизм обработки исключений предназначен только для событий, которые возникают в результате работы самой программы. Для программиста важно понимать разницу между ошибкой в программе и исключительной ситуаций.

*Ошибка в программе* допускается программистом при ее разработке. Например, вместо операции сравнения (==) используется операция присваивания (=). Программист должен исправить подобные ошибки на этапе отладки программы. Использование механизма обработки исключений не является защитой от ошибок данного вида.

Даже если программист исправил все свои ошибки в программе, он может столкнуться с непредсказуемыми и неотвратимыми проблемами — *исключительными ситуациями*. Например, нехваткой доступной памяти или попыткой открыть несуществующий файл. Исключительные ситуации программист предвидеть не может, но он может отреагировать на них так, чтобы они не привели к краху программы. Для обработки исключительных ситуаций в C++ используется специальный механизм обработки исключений.

### 13.1. Механизм обработки исключений

Механизм обработки исключений разделяет вычислительный процесс на две части — обнаружение исключительной ситуации и ее обработку. Рассмотрим данный механизм более подробно.

Обработка исключения начинается с появления ошибки. Функция, в которой она возникла, генерирует исключение с помощью оператора `throw` с параметром, определяющим тип исключения. Параметр может быть константой, переменной или объектом и используется для передачи информации об исключении его обработчику. Например:

```
throw 1; // параметр - целочисленная константа  
throw "Ошибка: деление на ноль"; // параметр - строковая константа  
throw k; // параметр - переменная
```

Сгенерированное исключение нужно перехватить и обработать. Перехват исключений и проверка возникновения исключения выполняются с помощью оператора **try**, с которым неразрывно связаны один или несколько блоков обработки исключений — **catch**. Оператор **try** объявляет в любом месте программы контролируемый блок, который имеет следующий вид:

```
try // контролируемый блок  
{ ... }
```

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в контролируемый блок. Контролируемый блок, помимо функций контроля, обладает функциями обычного блока: все идентификаторы, объявленные внутри него, являются локальными в этом блоке и не видны вне его.

За блоком **try** следуют один или несколько блоков **catch** — блоков обработки исключений. Формат записи блока **catch** следующий:

```
catch (спецификация_параметра_исключения) // блок обработки  
{ ... }
```

Спецификация параметра исключения может иметь три формы:

- 1) (тип имя);
- 2) (тип);
- 3) (...).

Первый вариант спецификации означает, что объект-исключение передается в блок обработки для использования, например, для вывода информации об аварийной ошибке. При этом объект-исключение может передаваться в блок **catch** по значению, по ссылке или по указателю. Например:

```
catch (int exception) { ... }           // по значению  
catch (int & exception) { ... }          // по ссылке  
catch (const int & exception) { ... }    // по константной ссылке  
catch (int *exception) { ... }          // по указателю
```

Вторая форма спецификации предназначена для обработки исключения конкретного типа, но без передачи объекта-исключения в блок **catch**. Например:

```
catch (int) { ... } // обработка исключений целого типа  
catch (double) { ... } // обработка исключений вещественного типа
```

Третий вариант спецификации используется для обработки исключений всех типов.

Работа конструкции *try-catch* напоминает работу оператора *switch*. Если в блоке **try** сгенерировано исключение, то начинается сравнение

типа сгенерированного исключения с типами параметров блоков **catch**. Выполняется тот блок **catch**, тип которого совпал с типом исключения. Если блок **catch** с заданным типом не найден, то выполняется блок **catch** с многоточием. Если же такого блока нет, то выполняется поиск в вызывающей функции, который продолжается до функции *main*. Если же и там не обнаруживается нужного блока **catch**, то вызывается стандартная функция завершения *terminate()*, которая вызывает функцию *abort()*. Таким образом, очень важен порядок записей блоков **catch** — если в качестве первого блока указать блок с параметром-многоточием, то такой обработчик будет обрабатывать все типы исключений и до остальных блоков **catch** просто не дойдет. Поэтому нужно усвоить следующее правило — блок **catch** с параметром-многоточием должен быть последним.

Следует отметить, что выход из блока **catch** выполняется одним из нескольких способов.

1. Выполняются все операторы блока **catch**, после чего управление передается оператору, расположенному после конструкции *try-catch*.

2. В блоке **catch** выполняется оператор **goto**, при этом разрешается переходить на любой оператор вне конструкции *try-catch* (внутрь контролируемого блока или в другой блок **catch** переход запрещен).

3. В блоке **catch** выполняется оператор **return**, после чего происходит нормальный выход из функции.

4. В блоке **catch** повторно генерируется другое исключение.

Рассмотрим следующий пример:

```
#include "iostream"
using namespace std;
int main()
{
    try
    {
        throw "сгенерировано исключение";
        cout << "действия после генерации исключения" << endl;
    }
    catch (int)
    {
        cout << "целочисленная ошибка" << endl;
    }
    catch (char* s)
    {
        cout << s << endl;
    }
    catch (...)
    {
        cout << "возникла какая-то ошибка" << endl;
    }
    cout << "конец программы" << endl; // 1
}
```

Когда генерируется исключение, выполнение программы останавливается и управление передается блоку **catch** соответствующего типа.

Этот блок никогда не возвращает управление в то место программы, где возникло исключение. Поэтому команды из блока `try`, расположенные ниже строки, в которой было сгенерировано исключение, никогда не будут выполнены. В нашем примере сгенерировано исключение строкового типа, поэтому управление будет передано блоку `catch (char *s)`, и на экран будет выведено сообщение "сгенерировано исключение". После чего управление будет передано строке с номером 1 и на экран будет выведено сообщение "конец программы".

## 13.2. Применение исключений на практике

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на возможную ошибку при вычислениях, а затем продолжить выполнение без прерывания работы программы. Например, нам нужно построить таблицу значений для функции вида  $y(x) = \frac{100}{x^2 - 1}$  на отрезке  $[a, b]$  с шагом  $h$ . Тогда программа может выглядеть следующим образом:

```
#include "iostream"
#include "iomanip"
using namespace std;
int main()
{
    double a, b, h, x, y;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "h = ";
    cin >> h;
    cout << left << setprecision(5);
    cout << setw(10) << 'x' << setw(10) << 'y' << endl;
    for (x = a; x <= b; x += h)
    {
        try
        {
            if (x == 1 || x == -1)
                throw "деление на ноль";
            y = 100.0 / (x * x - 1);
            cout << setw(10) << x << setw(10) << y << endl;
        }
        catch (char *s)
        {
            cout << setw(10) << x << s << endl;
        }
    }
    return 0;
}
```

Результат выполнения программы:

— входные данные:

```
a = -2
b = 2
h = 1
```

— выходные данные:

```
x   y
-2  33.33
-1  деление на ноль
0   -100
1   деление на ноль
2   33.33
```

В некоторых случаях исключительная ситуация может возникнуть во вспомогательной функции. Информацию об этом можно передать в вызывающую функцию с помощью повторной генерации исключения.

Например, построить таблицу значений для функции вида  $y(x) = \frac{100}{x^2 - 1}$  на отрезке  $[a, b]$  с шагом  $h$  с использованием вспомогательной функцией. Тогда программа будет выглядеть следующим образом:

```
#include "iostream"
#include "iomanip"
using namespace std;
double f(double x) // вспомогательная функция
{
    try
    {
        if (x == 1 || x == -1)
            throw "деление на ноль"; // генерация исключения
    }
    catch (char *s) // обработка исключения
    {
        // повторная генерация исключения, информация
        // о котором передается в вызывающую функцию
        throw s;
    }
    return 100.0 / (x * x - 1);
}

int main() // основная функция
{
    double a, b, h, x, y;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "h = ";
    cin >> h;
    cout << left << setprecision(5);
    cout << setw(10) << 'x' << setw(10) << 'y' << endl;
```

```
for (x = a; x <= b; x += h) {  
    try // пытаемся выполнить вспомогательную функцию  
    {  
        y = f(x);  
        cout << setw(10) << x << setw(10) << y << endl;  
    }  
    // обработка исключения, переданного из вспомогательной функции  
    catch (char *s)  
    {  
        cout << setw(10) << x << s << endl;  
    }  
}  
return 0;  
}
```

### Замечания

---

1. Результат выполнения данной программы будет соответствовать предыдущему примеру.
  2. Чтобы более подробно познакомиться с механизмом обработки исключительных ситуаций, рекомендуем воспользоваться дополнительной литературой.
- 

## Упражнения

Постройте таблицу значений функции  $y = f(x)$  для  $x \in [a, b]$  с шагом  $h$ . Если в некоторой точке  $x$  функция не определена, то выведите на экран сообщение об этом. Функции  $y = f(x)$  использовать из параграфа «Упражнения» к гл. 3, п. V.

### Замечание

---

При решении данной задачи использовать вспомогательную функцию  $f(x)$ , которая вычисляет значение  $y$ , а также проводить обработку возможных исключений.

---

# Глава 14

## КЛАССЫ И ОБЪЕКТЫ

### 14.1. Основные понятия

Определение класса выглядит следующим образом:

```
class <имя класса>
{
    <тело класса>
};
```

Таким образом, определение класса начинается с ключевого слова **class**, за которым следует идентификатор, обозначающий имя данного класса. Затем в фигурных скобках определяется тело класса. Завершается описание класса точкой с запятой.

В теле класса (которое может быть и пустым, тогда это *пустой класс*) определяют данные и операции. Данные и операции, являющиеся частью класса, называются **членами класса**. Данные также называются **членами-данными** (полями), а операции — **членами-функциями**.

Класс может содержать любое количество разделов, помеченных модификаторами доступа **private** (закрытый), **public** (открытый), **protected** (защищенный). Модификаторы доступа **private** и **public** указывают, будет ли член класса доступен вне объекта данного класса; модификатор **protected** будет рассмотрен позже. Модификатор доступа остается в силе до тех пор, пока в описании не встретится другой модификатор или не закончится описание класса. По умолчанию все члены класса автоматически считаются закрытыми.

Рассмотрим пример класса *MyClass*, который содержит поле *data* и две функции: *SetData*, которая задает значение поля *data*, и *ShowData*, которая отображает значение поля *data*.

```
class MyClass
{
private:
    int data;
public:
    void SetData (int d)
    {
        data = d;
    }
}
```

```
void ShowData()
{
    cout << "Data = " << data << endl;
}
};
```

Так как для компилятора класс — это пользовательский тип данных, то для работы с классом необходимо создать экземпляр класса (объект), аналогично тому, как создается (объявляется) переменная требуемого типа. Например:

```
int x;      // объявление целочисленной переменной
MyClass y; // создание экземпляра класса
```

Чтобы обратиться к члену класса, необходимо указать имя объекта, точку, а затем — имя функции или поля класса:

```
int main()
{
    // объявление объектов класса MyClass
    MyClass firstObject;
    MyClass secondObject;
    // вызов метода SetData для первого объекта
    firstObject.SetData(5);
    // вызов метода SetData для второго объекта
    secondObject.SetData(10);
    // вызов метода ShowData для первого объекта
    firstObject.ShowData();
    // вызов метода ShowData для второго объекта
    secondObject.ShowData();
    return 0;
}
```

Результат работы программы:

```
Data = 5
Data = 10
```

В данном примере мы объявили два объекта класса *MyClass*: *firstObject* и *secondObject*. Поскольку поле *data* класса *MyClass* помечено модификатором доступа **private**, невозможно обратиться к нему извне класса, в частности из функции *main*. Например, невозможно присвоить значение данному полю напрямую:

```
firstObject.data = 5; // Ошибка!!!
```

Вместо этого мы должны использовать метод *SetData*.

Возможность **сокрытия данных** является ключевой особенностью ООП. Этот термин понимается в том смысле, что данные, помеченные модификатором **private**, доступны только внутри класса. Скрытие данных позволяет избежать многих ошибок. Приведем пример. Пусть какое-то поле класса может принимать значения из интервала от 0 до 10. Если присвоить ему значение, меньшее 0 или большее 10, функ-

ции класса будут работать некорректно. В том случае, когда присвоение значения полю класса возможно только с помощью функции того же класса, данное ограничение будет заложено внутри функции. Например, при вводе некорректного значения оно будет считаться нулем, или пользователя попросят ввести значение еще раз. Если же доступ к этому полю будет осуществляться из любого другого места программы, то нет никакой гарантии, что ограничение будет соблюдаться.

Объекты класса разрешается определять в качестве полей другого класса. Объекты класса можно передавать в качестве аргументов любыми способами (по значению, по указателю, по ссылке) и получать как результат функции.

Итак, предположим, что мы определили класс. По завершении определения все члены класса должны быть известны. В результате объем памяти, необходимый для хранения объекта данного класса, тоже будет известен. Класс можно объявить, но не определять, указав только его имя. Например,

```
class myClass; // объявление класса
```

Такое объявление иногда называют предварительным. После объявления, но до определения данный класс является незавершенным типом, т.е. известно, что это класс, но неизвестно, что он содержит. Объект такого класса создать нельзя. Как правило, предварительное объявление используют, когда необходимо создать взаимозависящие классы.

Функции класса находятся в памяти в единственном экземпляре и используются всеми объектами одного класса совместно, поэтому необходимо обеспечить работу функций с полями именно того объекта, для которого они были вызваны. Для этого в любую функцию класса автоматически передается скрытый параметр **this**, в котором хранится ссылка на вызвавший функцию экземпляр класса.

В явном виде параметр **this** применяется для того, чтобы возвратить из функции ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра функции, например:

```
void SetData (int data)
{
    this->data = data;
}
```

## 14.2. Конструкторы

Для инициализации полей объекта класса *MyClass* мы использовали функцию *SetData*. Ее необходимо вызывать явно каждый раз при создании соответствующего объекта. Существует вероятность того, что программист забудет вызвать данную функцию, и поле не будет про-

инициализировано, что приведет к неправильной работе программы. Для решения данной проблемы используется конструктор.

**Конструктор** — это специальная член-функция, которая вызывается неявно каждый раз, когда создается новый объект класса. Конструктор предназначен для того, чтобы присвоить каждому члену-данному начальное значение. Он отличается от других членов-функций тем, что его имя совпадает с именем класса. Кроме того, конструкторы не имеют типа возвращаемого значения, но имеют список параметров (который может быть пуст) и тело. Каждый класс может иметь несколько конструкторов, которые должны отличаться друг от друга количеством или типом параметров. Параметрами конструктора являются значения, используемые для инициализации членов-данных класса при создании объекта.

Общий вид конструктора следующий:

```
<имя конструктора> (<список параметров>):
    <имя поля 1> (<инициализирующее значение для поля 1>), ...,
    <имя поля N> (<инициализирующее значение для поля N>
{
    <тело конструктора>
}
```

Для нашего класса можно определить следующий конструктор

```
MyClass():data(0)
{
}
```

В данном случае поле *data* будет инициализироваться значением 0. Тогда определение класса выглядит следующим образом:

```
class MyClass
{
private:
    int data;
public:
    MyClass() : data(0) {}
    void SetData(int data)
    {
        this->data = data;
    }
    void ShowData()
    {
        cout << "Data = " << data << endl;
    }
};
```

Работа с этим классом:

```
int main()
{
    MyClass firstObject;
    MyClass secondObject;
```

```
    firstObject.ShowData();
    secondObject.ShowData();
    return 0;
}
```

Результат работы программы:

```
Data = 0
Data = 0
```

Конструктор и другие члены-функции класса можно перегружать как обычные функции. Рассмотрим, например, конструктор класса *MyClass*, который будет инициализировать поле *data* заданным значением.

```
MyClass(int d) : data(d)
{
}
```

В данном случае используется конструктор с параметром *d*, который используется для инициализации поля *data*. Допустимо, чтобы имя параметра совпадало с именем поля. Например:

```
MyClass(int data) : data(data)
{
}
```

Теперь определение класса выглядит следующим образом:

```
class MyClass
{
private:
    int data;
public:
    MyClass() : data(0) // конструктор 1
    {
    }
    MyClass(int data) : data(data) // конструктор 2
    {
    }
    void SetData(int data)
    {
        this->data = data;
    }
    void ShowData()
    {
        cout << "Data = " << data << endl;
    }
};

Работа с данным классом:
int main()
{
    MyClass firstObject;
    MyClass secondObject(10);
```

```
firstObject.ShowData();
secondObject.ShowData();
return 0;
}
```

Результат работы программы:

```
Data = 0
Data = 10
```

#### Замечание

Если в классе не определен ни один конструктор, компилятор создает так называемый синтезируемый стандартный конструктор. Синтезируемый стандартный конструктор инициализирует члены-данные по тем же правилам, что и обычные переменные, а именно:

- члены-данные, являющиеся объектами класса, инициализируются их собственным конструктором;
- члены-данные встроенного или составного типа, такие как указатели или массивы, инициализируются только для тех объектов, которые определены в глобальной области видимости;
- объекты встроенных типов (например, типа *int*) или составных типов, определенные в локальной области видимости, остаются неинициализированными, следовательно, использовать их для чего-либо отличного от присвоения (или ввода) значения нельзя.

## 14.3. Деструкторы

Как мы уже видели, существует особая функция класса — конструктор, которая вызывается автоматически при создании объекта. Существует еще одна функция — деструктор, которая автоматически вызывается при уничтожении объекта. Напомним, что уничтожение локальных объектов происходит при выходе из блока, а для объектов, которые создаются посредством операции **new**, — при вызове операции **delete**.

Деструктор имеет имя класса, перед которым стоит значок  $\sim$  (тильда). Деструктор не возвращает результат и не имеет параметров. Деструктор в классе может быть один — перегрузка деструкторов не разрешается. Если деструктор не определен явно, он создается системой автоматически.

Рассмотрим пример деструктора.

```
~ MyClass()
{
    cout << "Delete: data = " << data;
}
```

Выполним предыдущую реализацию функции *main* еще раз.

Результат работы программы:

```
Data = 0
Data = 10
Delete: data = 10
Delete: data = 0
```

### Задание

Попробуйте объяснить, почему деструкторы были вызваны именно в таком порядке.

Обычно деструктор явно используется только тогда, когда при уничтожении необходимо выполнить некоторые специальные действия, например закрыть файл, открытый в конструкторе, или освободить память, выделенную динамическим образом под некоторый член данных. Если же класс не используется для обеспечения доступа к какому-либо системному ресурсу, использование деструкторов настоятельно не рекомендуется.

## 14.4. Статические члены класса

Зачастую случается такая ситуация, когда объект класса должен располагать информацией, сколько еще объектов этого класса существует на данный момент. Например, если объектами являются участники соревнований, необходимо, чтобы каждый участник знал о том, сколько всего человек участвуют в соревнованиях. В этом случае необходимо использовать статическую переменную — член класса. Эта переменная будет видна всем объектам данного класса и для всех будет иметь одинаковое значение.

Статическое поле описывается с помощью ключевого слова **static**. Статическое поле существует даже в том случае, когда не создано ни одного объекта класса. Поэтому статические поля нельзя инициализировать в конструкторе. Определение начального значения статических полей происходит вне класса (после определения класса) и выглядит следующим образом:

```
<тип поля> <имя класса>::<имя поля> = <инициализатор>;
```

или

```
<тип поля> <имя класса>::<имя поля> ;
```

Рассмотрим пример использования класса *MyClass*, в котором определим статическое поле *count* для подсчета количества объектов данного класса:

```
#include "iostream"
using namespace std;
class MyClass
```

```

{
private:
    int data;
    static int count; // объявление статического поля
public:
    MyClass() : data(0)
    {
        count++; // увеличение значения статического поля
    }
    MyClass(int data) : data(data)
    {
        count++; // увеличение значения статического поля
    }
    int GetCount() // функция, возвращающая значение статического поля
    {
        return count;
    }
    void SetData(int data)
    {
        this->data = data;
    }
    void ShowData()
    {
        cout << "Data = " << data << endl;
    }
};
int MyClass::count = 0; // определение начального значения
                        // статического поля
int main()
{
    MyClass firstObject;
    MyClass secondObject(10);
    cout << "Count = " << firstObject.GetCount() << endl;
    cout << "Count = " << secondObject.GetCount() << endl;
    return 0;
}

```

Результат работы программы:

```

Count = 2
Count = 2

```

Таким образом, значение счетчика одинаково для любого объекта данного класса. На самом деле, начальное значение счетчику присвоено до создания экземпляров класса, а каждый раз при объявлении объекта данного класса неявно вызывается конструктор, который увеличивает это значение на единицу. В данном примере было объявлено два объекта *MyClass*, следовательно, конструктор класса вызывался два раза, именно поэтому значение счетчика равно двум.

## 14.5. Перегрузка операций

Перегрузка операций в C++ позволяет переопределить большинство операций так, чтобы при использовании их объектами конкретного

класса выполнялись действия, отличные от стандартных. Это дает возможность применять объекты собственных типов данных в составе выражений, например:

```
MyClass x, y, z;  
// используется операция сложения, переопределенная для класса  
MyClass  
z = x + y;
```

Перегрузка операций обычно применяется для классов, для которых семантика операций делает программу более понятной. Если назначение операции интуитивно непонятно, перегружать такую операцию не рекомендуется.

Начнем изучение перегрузки с **унарных операций**. Унарные операции имеют только один operand. Примерами унарных операций могут служить операции инкремента (++) и декремента (--).

Синтаксис перегрузки унарной операции:

```
<тип результата> operator <унарная_операция> ()  
{  
    <реализация перегрузки>  
}
```

Добавим в раздел **public** класса *MyClass* следующую реализацию перегрузки операции инкремента.

```
void operator ++()  
{  
    ++data;  
}
```

Продемонстрируем работу данной операции.

```
int main()  
{  
    MyClass firstObject;  
    firstObject.ShowData();  
    ++firstObject;  
    firstObject.ShowData();  
    return 0;  
}
```

Результат работы программы:

```
Data = 0  
Data = 1
```

Однако при попытке выполнить следующие команды:

```
MyClass firstObject;  
MyClass secondObject = ++firstObject; // Ошибка!!!
```

компилятор выдаст сообщение об ошибке «невозможно преобразовать *void* в *MyClass*». Почему? Потому, что мы определили тип резуль-

тата при перегрузке операции инкремента как `void`. Чтобы иметь возможность использовать предложенную нами версию перегрузки инкремента, нам необходимо правильно определить тип возвращаемого значения. Это можно сделать следующим образом:

```
MyClass operator ++()
{
    return MyClass(++data);
}
```

Здесь функция `operator++` вначале увеличивает на единицу свое поле `data`, а затем создает новый объект класса `MyClass`, используя новое значение поля `data` для инициализации. В качестве результата функция `operator++` возвращает ссылку на новый объект. Продемонстрируем работу перегрузки операции на примере.

```
int main()
{
    MyClass firstObject;
    MyClass secondObject = ++firstObject;
    ++secondObject;
    firstObject.ShowData();
    secondObject.ShowData();
    return 0;
}
```

Результат работы программы:

```
Data = 1
Data = 2
```

### Задание

Мы определяли операцию инкремента, используя префиксную запись. Самостоятельно разработайте вариант для постфиксной записи инкремента.

Аналогичным образом можно перегрузить **бинарные операции**, например операцию сложения:

```
MyClass operator + (MyClass temp)
{
    return MyClass(data + temp.data); // 1
}
```

Рассмотрим реализацию данной перегрузки на примере.

```
int main()
{
    MyClass firstObject(1);
    MyClass secondObject(2);
    MyClass thirdObject = firstObject + secondObject;
    thirdObject.ShowData();
    return 0;
}
```

Результат работы программы:

```
Data = 3
```

Объявление функции *operator+()* в классе *MyClass* выглядит следующим образом:

```
MyClass operator + (MyClass temp)
```

Эта функция возвращает значение типа *MyClass* и принимает один аргумент типа *MyClass*.

В выражении

```
thirdObject = firstObject + secondObject;
```

важно понимать, к каким объектам будут относиться аргументы и возвращаемые значения. Когда компилятор встречает это выражение, то он просматривает типы аргументов. Обнаружив только аргументы типа *MyClass*, он выполняет операции выражения, используя функцию класса *MyClass operator+()* по правилу: объект, стоящий с левой стороны от знака операции (в нашем случае *firstObject*), вызывает функцию перегрузки бинарной операции *+*. Объект, стоящий справа от знака операции, передается в функцию в качестве аргумента (в нашем случае *secondObject*). Операция возвращает значение, которое записывается в *thirdObject*.

В функции *operator+()* (строка 1) мы имеем прямой доступ к левому операнду, используя поле *data*, так как именно этот объект вызывает функцию. К правому операнду мы имеем доступ как к аргументу функции, т.е. *temp.data*. Таким образом, перегруженной операции требуется на один аргумент меньше, чем количество операндов, так как один из операндов является объектом,зывающим функцию перегрузки. Именно поэтому для функций перегрузки унарных операций не нужны аргументы.

## 14.6. Пример простого класса

Создать класс *Point*, который содержит следующие члены класса.

1. Поля *int x, y*;
  2. Конструкторы, позволяющие создать экземпляр класса:
    - с нулевыми координатами;
    - с заданными координатами.
  3. Функции, позволяющие:
    - определить общее количество точек;
    - установить координаты точки;
    - вывести координаты точки на экран;
    - рассчитать расстояние от начала координат до точки.
- Кроме того, необходимо реализовать перегрузку:
- операции, которая одновременно уменьшает значение полей *x* и *y* на единицу;

- операции «бинарный +», которая позволяет сложить координаты двух точек, получив новую точку (вариант 1);
- операции «бинарный +», которая позволяет увеличить координаты точки на заданное значение (вариант 2).

Реализация и использование класса:

```
#include <iostream>
#include <math.h>
#include <string>
using namespace std;
class Point
{
private:
    // координаты точки
    int x;
    int y;
    static int count; // количество объектов
public:
    // конструктор с нулевыми координатами
    Point() : x(0), y(0) { count++; }
    // конструктор с заданными координатами
    Point(int x, int y) : x(x), y(y) { count++; }
    // функция позволяет определить количество объектов типа Point
    int GetCount() { return count; }
    // функция позволяет установить новые координаты точки
    void SetXY(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
    // функция позволяет вывести на экран информацию о точке
    void Show(string info)
    {
        cout << "Point " << info << ": ( " << x << ", " << y << " )"
            << endl;
    }
    // функция позволяет вычислить расстояние
    // от начала координат до точки
    double GetDistance()
    {
        return sqrt((double)(x * x + y * y));
    }
    // перегрузка операции ++
    Point operator++()
    {
        return Point(++x, ++y);
    }
    // перегрузка операции --
    Point operator--()
    {
        return Point(--x, --y);
    }
    // Вариант 1 перегрузки операции +
    Point operator+(Point temp)
```

```

{
    return Point(x + temp.x, y + temp.y);
}
// Вариант 2 перегрузки операции +
Point operator+(int a)
{
    return Point(x + a, y + a); }
};

int Point::count = 0; // инициализация статического поля
int main()
{
    Point pointA; // создаем точку A с нулевыми координатами
    pointA.Show("A");
    cout << "Distance: " << pointA.GetDistance() << endl << endl;
    pointA.SetXY(-3, 4); // устанавливаем новые координаты точки A
    pointA.Show("A");
    cout << "Distance: " << pointA.GetDistance() << endl << endl;
    Point pointB(3, 4); // создаем точку B с заданными координатами
    pointB.Show("B");
    cout << "Distance: " << pointB.GetDistance() << endl << endl;
    --pointB; // демонстрация перегрузки операции --
    pointB.Show("B");
    cout << "Distance: " << pointB.GetDistance() << endl << endl;
    // демонстрация варианта 1 перегрузки операции +
    Point pointC = pointA + pointB;
    pointC.Show("C");
    cout << "Distance: " << pointD.GetDistance() << endl << endl;
    // демонстрация варианта 2 перегрузки операции +
    Point pointD = pointC + 3;
    pointD.Show("D");
    cout << "Distance: " << pointD.GetDistance() << endl << endl;
    // выводим общее количество точек
    cout << "Count of point: " << pointD.GetCount();
    return 0;
}

```

Результат работы программы:

```

Point A: (0, 0)
Distance: 0
Point A: (-3, 4)
Distance: 5
Point B: (3, 4)
Distance: 5
Point B: (2, 3)
Distance: 3.60555
Point C: (-1, 7)
Distance: 7.07107
Point D: (2, 10)
Distance: 10.198
Count of Point: 5

```

### Задание

Как мы видим из примера, у нас существуют четыре точки: A, B, C, D. Подумайте и объясните, почему функция *GetCount()* выдала в качестве общего

количества существующих объектов класса *Point* значение 5. Исправьте данную ошибку.

---

## Упражнения

### Замечание

В каждом задании класс должен содержать конструкторы (без параметров и конструктор инициализации), а также статическое поле для подсчета количества экземпляров класса. Работу классов продемонстрируйте на примерах.

---

1. Создать класс *Triangle*, содержащий следующие члены класса:
  - 1) поля *int a, b, c*;
  - 2) функции, позволяющие:
    - вывести на экран информацию о треугольнике,
    - рассчитать периметр треугольника,
    - рассчитать площадь треугольника,
    - установить длины сторон треугольника,
    - установить, существует ли треугольник с данными длинами сторон.

Кроме того, необходимо реализовать перегрузку:

  - операции *++* (*--*): одновременно увеличивает (уменьшает) значение полей *a*, *b* и *c* на единицу;
  - операции *\**: умножает поля *a*, *b* и *c* на заданный скаляр.
2. Создать класс *Rectangle*, содержащий следующие члены класса:
  - 1) поля *int a, b*;
  - 2) функции, позволяющие:
    - вывести на экран информацию о прямоугольнике,
    - рассчитать периметр прямоугольника,
    - рассчитать площадь прямоугольника,
    - установить длины сторон прямоугольника,
    - установить, является ли данный прямоугольник квадратом.

Кроме того, необходимо реализовать перегрузку:

  - операции *++* (*--*): одновременно увеличивает (уменьшает) значение полей *a* и *b*;
  - операции *\**: умножает поля *a* и *b* на заданный скаляр.
3. Создать класс *Money*, содержащий следующие члены класса:
  - 1) поля:
    - *int nominal* — номинал купюры,
    - *int amount* — количество купюр;
  - 2) функции, позволяющие:
    - вывести номинал и количество купюр на экран,
    - определить, хватит ли денежных средств на покупку товара на сумму *X* руб.,
      - определить, сколько штук товара стоимостью *Y* руб. можно купить на имеющиеся денежные средства,
      - установить значение полей,
      - рассчитать сумму денег.

Кроме того, необходимо реализовать перегрузку:

  - операции *++* (*--*): увеличивает (уменьшает) значение поля *amount*;
  - операции *+*: добавляет к значению поля *amount* значение скаляра.

4. Создать класс *Rational* для работы с рациональными дробями, содержащий следующие члены класса:

- 1) поля:
  - *int numerator* — числитель дроби,
  - *int denominator* — знаменатель дроби;
- 2) функции, позволяющие:
  - вывести на экран дробь,
  - задать новые значения числителя и знаменателя дроби,
  - провести сокращение дроби,
  - сравнить две дроби.

Кроме того, необходимо реализовать перегрузку операций сложения, умножения, вычитания и деления дробей.

5. Создать класс *Complex* для работы с комплексными числами (комплексное число определяется парой вещественных чисел: действительная часть, мнимая часть), содержащий следующие члены класса:

- 1) поля:
  - *double realPart* — действительная часть,
  - *double imaginaryPart* — мнимая часть;
- 2) функции, позволяющие:
  - вывести на экран комплексное число,
  - создать сопряженное число,
  - сравнить два числа на равенство.

Кроме того, необходимо реализовать перегрузку операций сложения, умножения, вычитания и деления комплексных чисел.

6. Создать класс *Vector* для работы с векторами на плоскости, содержащий следующие члены класса:

- 1) поля *int x, y*;
- 2) функции, позволяющие:
  - вывести на экран вектор,
  - вычислить длину вектора,
  - сравнить два вектора на равенство.

Кроме того, необходимо реализовать перегрузку операций сложения, вычитания, скалярного и векторного произведения.

7. Создать класс *Vector3D* для работы с векторами в пространстве, содержащий следующие члены класса:

- 1) поля *int x, y, z*;
- 2) функции, позволяющие:
  - вывести на экран вектор,
  - вычислить длину вектора,
  - сравнить два вектора на равенство.

Кроме того, необходимо реализовать перегрузку операций сложения, вычитания, скалярного и векторного произведения.

8. Создать класс для работы с одномерным массивом целых чисел. Разработать следующие члены класса:

- 1) поле *int [] IntArray*;
- 2) функции, позволяющие:
  - ввести элементы массива с клавиатуры,
  - вывести элементы массива на экран,
  - отсортировать элементы массива в порядке возрастания,
  - узнать размерность массива,
  - вычислить сумму элементов в массиве.

Кроме того, необходимо реализовать перегрузку:

- операции `++` (`--`), которая позволяет одновременно увеличить (уменьшить) значение всех элементов массива на единицу;
- операции `*`, которая позволяет домножить все элементы массива на скаляр.

9. Создать класс для работы с двумерным массивом целых чисел. Разработать следующие члены класса:

- 1) поле `int [][] intArray;`
- 2) функции, позволяющие:
  - ввести элементы массива с клавиатуры,
  - вывести элементы массива на экран,
  - вычислить сумму элементов  $i$ -го столбца,
  - вычислить количество нулевых элементов в массиве,
  - установить значение всех элементов главной диагонали массива, равное скаляру.

Кроме того, необходимо реализовать перегрузку:

- операции `++` (`--`), которая позволяет одновременно увеличить (уменьшить) значение всех элементов массива на единицу;
- операции `+`, которая позволяет сложить два массива соответствующих размерностей.

10. Создать класс для работы со строками. Разработать следующие члены класса:

- 1) поле `string line;`
- 2) функции, позволяющие:
  - подсчитать количество цифр в строке,
  - выводить на экран все символы строки, встречающиеся в ней ровно один раз,
  - вывести на экран самую длинную последовательность повторяющихся символов в строке,
  - узнать общее количество символов в строке,
  - сравнить две строки на равенство.

Кроме того, необходимо реализовать перегрузку:

- операции унарный `!`, которая возвращает значение `true`, если строка не пустая, иначе `false`;
- операции `+`, которая позволяет реализовать операцию слияния двух строк, т.е. получить новую строку, добавив в конец первой начало второй.

11\*. Создать класс `Time` с полями `hours`, `minutes`, `seconds`, представляющими собой часы, минуты и секунды. Данный класс должен позволять выводить информацию о текущем времени, вычислять время через заданное количество часов, минут и секунд, а также вычислять время, прошедшее между двумя заданными моментами. Детальную структуру класса продумайте самостоятельно.

12\*. Создать класс `Data` с полями `year`, `month`, `day`, представляющими собой год, месяц и день. Данный класс должен позволять выводить информацию о текущей дате, вычислять дату следующего дня, дату предыдущего дня, дату дня через заданное количество лет, месяцев и дней, а также вычислять временной отрезок (в годах, месяцах и днях) между двумя заданными датами. Детальную структуру класса продумайте самостоятельно.

# Глава 15

## НАСЛЕДОВАНИЕ

### 15.1. Основные понятия

**Наследование** — это процесс создания новых классов (они называются производными классами или наследниками) на основе уже существующих классов (они называются базовыми). Производный класс наследует все члены базового класса, но также может их изменять и добавлять новые. Наследование позволяет использовать существующий код несколько раз.

Описание производного класса выглядит следующим образом:

```
class <имя производного класса> [<модификатор доступа>] <имя базового
класса>
{
    <тело производного класса>
}
```

При создании одиночных классов мы использовали два модификатора доступа, **public** и **private**. При наследовании применяется еще один модификатор — **protected** (защищенный). Защищенные элементы класса доступны только прямым наследникам и никому другому.

Рассмотрим простое наследование классов на примере геометрических фигур. В качестве базового класса создадим класс *Point* (точка на плоскости), в качестве производного класса от *Point* — класс *PointSpace* (точка в пространстве):

```
#include <iostream>
#include <string>
using namespace std;
class Point // базовый класс
{
public:
    int x;
    int y;
    void Show(string info)
    {
        cout << "Point " << info << ": ( " << x << ", " << y
           << " )" << endl;
    }
};
class PointSpace : public Point // производный класс
{
```

```

public:
    int z;
    void Show(string info)
    {
        cout << "Point " << info << ": ( " << x << ", " << y
            << ", " << z << " )" << endl;
    }
};

int main()
{
    Point pointA;
    pointA.x = 0;
    pointA.y = 0;
    pointA.Show("A");
    PointSpace pointB;
    pointB.x = 1;
    pointB.y = 1;
    pointB.z = 1;
    pointB.Show("B");
    return 0;
}

```

Результат работы программы:

```

Point A: (0, 0)
Point B: (1, 1, 1)

```

Таким образом, класс *PointSpace* унаследовал от класса *Point* поля *x*, *y*. В класс *PointSpace* было добавлено новое поле *z*, также в данном классе была переопределена функция *Show*.

## 15.2. Наследование конструкторов

В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы. Если в базовом классе нет конструкторов, то в производном классе конструктор можно не описывать — он будет создан неявным образом. Если же в базовом классе конструктор задается явным образом, то производный класс должен иметь собственный конструктор, в котором явно вызывается конструктор базового класса. При этом конструктор базового класса создает часть объекта, соответствующую базовому классу, а конструктор производного класса — часть объекта, соответствующую производному классу.

В предыдущем примере классы создавались за счет автоматического вызова средствами C++ конструктора по умолчанию. Теперь рассмотрим механизм наследования конструкторов на следующем примере:

```

#include <iostream>
#include <string>
using namespace std;
class Point // базовый класс
{

```

```

protected:
    int x;
    int y;
public:
    Point(): x(0), y(0) // конструктор без параметров
    {}
    Point(int x, int y): x(x), y(y) // конструктор с параметрами
    {}
    void Show(string info)
    {
        cout << "Point " << info << ": ( " << x << ", "
            << y << " )" << endl;
    }
};

class PointSpace : public Point // производный класс
{
protected:
    int z;
public:
    // наследование конструктора без параметров
    PointSpace() : Point(), z(0)
    {}
    // наследование конструктора с параметром
    PointSpace(int x, int y, int z) : Point(x, y), z(z)
    {}
    void Show(string info)
    {
        cout << "Point " << info << ": ( " << x << ", " << y
            << ", " << z << " )" << endl;
    }
};
int main()
{
    Point pointA;
    pointA.Show("A");
    Point pointB(1, 1);
    pointB.Show("B");
    PointSpace pointC;
    pointC.Show("C");
    PointSpace pointD(2, 2, 2);
    pointD.Show("D");
    return 0;
}

```

Результат работы программы:

```

Point A: (0, 0)
Point B: (1, 1)
Point C: (0, 0, 0)
Point D: (2, 2, 2)

```

### 15.3. Виртуальные функции

В языке C++ допустимо использовать указатель базового класса на объект производного класса, например, следующим образом:

```
Point *arrayPoint[4];
arrayPoint[0] = new Point(1, 1);
arrayPoint[1] = new PointSpace(2, 2, 2);
arrayPoint[2] = new PointSpace(3, 3, 3);
arrayPoint[3] = new Point(4, 4);
for (int i = 0; i < 4; i++)
{
    arrayPoint[i]->Show(""); // 1
}
```

### Замечание

---

В данном примере мы используем динамическое распределение памяти, поэтому при обращении к члену-класса *Show* вместо символа «.» следует использовать символ «->» (см. строку 1). Также после завершения работы с созданной структурой *arrayPoint* необходимо освободить выделенную память. Это можно сделать следующим образом:

```
for (int i = 0; i < 4; i++)
{
    delete arrayPoint[i];
}
```

---

В результате выполнения данного фрагмента программы мы ожидаем, что на экран будет выведено

```
Point: (1, 1)
Point: (2, 2, 2)
Point: (3, 3, 3)
Point: (4, 4)
```

Вместо этого мы получим

```
Point: (1, 1)
Point: (2, 2)
Point: (3, 3)
Point: (4, 4)
```

Дело в том, что все члены-функции класса заносятся в специальную таблицу функций. Это дает возможность хранить единственный экземпляр каждой функции и использовать его всем объектам одного класса совместно. Это решение вполне логично, так как если у нас есть 1000 объектов типа *Point*, то 1000 раз дублировать код функции *Show* нет никакого смысла. Для каждого класса строится собственная таблица функций. В нашем случае будет построено две таблицы — для класса *Point* и класса *PointSpace*.

В случае установки указателя типа *Point* на объект *PointSpace* возникает конфликт между двумя таблицами функций — базового класса и класса-потомка. Так как потомков у базового класса может быть много (в том числе потомков третьего и более высоких уровней), то поиск по всем возможным таблицам требуемой функции является

достаточно длительным процессом. Какая именно функция будет вызываться, определяется механизмом связывания вызывающего объекта с вызываемой функцией.

В C++ все функции по умолчанию имеют раннее связывание, т.е. компилятор и компоновщик решают, какая именно функция должна быть вызвана для заданного объекта, еще до запуска программы. В нашем случае *arrayPoint* объявлен типом *Point*, поэтому механизм раннего связывания для всех элементов данного массива вызывает функции класса *Point*.

Избежать подобную проблему можно с помощью использования виртуальных функций. Виртуальная функция — это функция, которая объявлена в базовом классе с использованием ключевого слова **virtual**. Ключевое слово **virtual** достаточно написать один раз — при объявлении функции базового класса. Далее во всех производных классах данная функция будет считаться виртуальной.

#### Замечание

Виртуальными не могут быть только конструкторы и статические функции. Подумайте, почему.

Использование виртуальных функций позволяет реализовать механизм позднего связывания, когда требуемая реализация функции выбирается на этапе выполнения программы. Рассмотрим, как реализуется механизм позднего связывания.

Когда в базовом классе объявляется хотя бы одна виртуальная функция, для всех полиморфных классов (базового и всех его производных) создается единая таблица виртуальных функций.

Помимо создания таблицы виртуальных функций в базовом классе создается специальный скрытый указатель на данную таблицу. Этот указатель наследуется всеми производными классами. При вызове виртуальной функции компилятор обращает внимание не на тип объекта, а осуществляет поиск требуемой реализации функции по таблице виртуальных функций.

Вернемся к нашему примеру. Объявим функцию *Show* в базовом классе *Point* как виртуальную функцию:

```
virtual void Show(string info)
{
    cout << "Point " << info << ": ( " << x << ", " << y
        << " )" << endl;
}
```

Тогда при выполнении фрагмента программы, приведенного в начале данного пункта, мы получим правильный результат:

```
Point: (1, 1)
Point: (2, 2, 2)
Point: (3, 3, 3)
Point: (4, 4)
```

## 15.4. Абстрактные классы и чисто виртуальные функции

Иногда полезно создать базовый класс, определяющий только своего рода «пустой бланк», который унаследуют все производные классы, причем каждый из них заполнит этот «бланк» собственной информацией. Такой класс определяет структуру функций, а их реализации будут предложены позже. Подобная ситуация может возникнуть в случае многоуровневого наследования, когда базовый класс не может предложить реализацию запланированных функций. Решить данную проблему можно с помощью абстрактных классов и чисто виртуальных функций.

Виртуальная функция, не имеющая тела, называется *чистой виртуальной функцией* и определяется следующим образом:

```
virtual <тип функции> <имя функции> (<список параметров>) = 0;
```

Класс, в котором есть хотя бы одна чистая виртуальная функция, называется абстрактным классом. Объекты абстрактного класса создавать запрещено. Кроме того, при передаче параметра в функцию невозможно передать объект абстрактного класса по значению, так как нельзя создать его копию.

При наследовании абстрактность сохраняется. Если класс-наследник не реализует унаследованную чистую виртуальную функцию, то он тоже является абстрактным. Часто абстрактные классы являются вершиной иерархии классов.

В качестве примера рассмотрим трехуровневую иерархию объектов, приведенную на рис. 15.1.

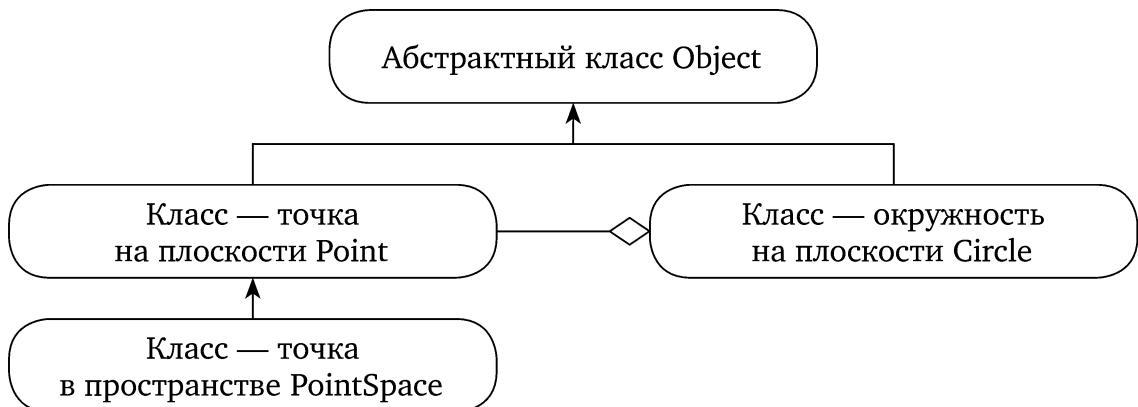


Рис. 15.1. Иерархия объектов

Стрелки с окончанием в виде черного треугольника показывают направление наследования (от производного класса к базовому), а стрелки с белым ромбом на конце — агрегацию. В данном случае агрегация говорит о том, что поле класса *Circle* будет являться указателем на объект класса *Point*, так как центр окружности является точкой.

Рассмотрим реализацию данной иерархии объектов.

```
#include <iostream>
#include <string>
using namespace std;
class Object // Абстрактный класс
{
public:
    virtual void Show() = 0;
    virtual double GetDistance() = 0;
};
class Point : public Object // класс – точка на плоскости
{
protected:
    int x;
    int y;
public:
    Point() : x(0), y(0)
    {}
    Point(int x, int y) : x(x), y(y)
    {}
    void Show()
    {
        cout << "Point : ( " << x << ", " << y << " )" << endl;
    }
    double GetDistance()
    {
        return sqrt((double)(x * x + y * y));
    }
    int GetX() { return x; }
    int GetY() { return y; }
};

class PointSpace : public Point // класс – точка в пространстве
{
protected:
    int z;
public:
    PointSpace() : Point(), z(0)
    {}
    PointSpace(int x, int y, int z) : Point(x, y), z(z)
    {}
    void Show()
    {
        cout << "Point: ( " << x << ", " << y << ", "
               << z << " )" << endl;
    }
    double GetDistance()
    {
        return sqrt((double)(x * x + y * y + z * z));
    }
    int GetZ() { return z; }
};
class Circle : public Object // класс – окружность на плоскости
{
protected:
```

```

Point *a; // проявление агрегации: поле класса Circle является
          // указателем на объект класса Point
    int radius;
public:
    Circle() : radius(0)
    {
        a = new Point(); // неявная инициализация центра окружности
    }
    Circle(int x, int y, int radius) : radius(radius)
    {
        a = new Point(x, y); // явная инициализация центра окружности
    }
    void Show()
    {
        cout << "Circle: center (" << a->GetX() << ", "
              << a->GetY() << "), radius " << radius << endl;
    }
    double GetDistance()
    {
        return sqrt(
            (double)(a->GetX() * a->GetX() + a->GetY() * a->GetY()));
    }
    ~Circle()
    {
        delete a; // освобождаем память, связанную с указателем a
    }
};

int main()
{
    Object *objects[6]; // массив указателей
    objects[0] = new Point(1, 1);
    objects[1] = new PointSpace(2, 2, 2);
    objects[2] = new Circle(0, 0, 3);
    objects[3] = new Circle(4, 3, 10);
    objects[4] = new PointSpace(-1, 2, -1);
    objects[5] = new Point(0, 2);
    for (int i = 0; i < 6; i++)
    {
        objects[i]->Show();
        cout << "Distance: " << objects[i]->GetDistance() << endl;
    }
    for (int i = 0; i < 6; i++)
    {
        delete objects[i];
    }
    return 0;
}

```

Результат работы программы:

```

Point: (1, 1)
Distance: 1.41421

```

```

Point: (2, 2, 2)
Distance: 3.4641

```

```
Circle: center (0, 0), radius 3
Distance: 0
Circle: center (4, 3), radius 10
Distance: 5
Point: (-1, 2, 1)
Distance: 2.44949
Point: (0, 2)
Distance: 2
```

## Упражнения

### Замечание

Перед выполнением задания продумайте и опишите полную структуру классов и их взаимосвязь.

1. Создать абстрактный класс *Figure* с функциями вычисления площади и периметра, а также функцией, выводящей информацию о фигуре на экран.

Создать производные классы: *Rectangle* (прямоугольник), *Circle* (круг), *Triangle* (треугольник).

Создать массив  $n$  фигур и вывести полную информацию о фигурах на экран.

2. Создать абстрактный класс *Function* с функциями вычисления значения по формуле  $y = f(x)$  в заданной точке, а также функцией, выводящей информацию о виде функции на экран.

Создать производные классы: *Line* ( $y = ax + b$ ), *Kub* ( $y = ax^2 + bx + c$ ), *Hyperbola* ( $y = a/x$ ).

Создать массив  $n$  функций и вывести полную информацию о значении данных функций в точке  $x$ .

3. Создать абстрактный класс *Edition* с функциями, позволяющими вывести на экран информацию об издании, а также определить соответствие издания критерию поиска.

Создать производные классы: *Book* (название, фамилия автора, год издания, издательство), *Article* (название, фамилия автора, название журнала, его номер и год издания), *OnlineResource* (название, фамилия автора, ссылка, аннотация).

Создать каталог (массив) из  $n$  изданий, вывести полную информацию из каталога, а также организовать поиск изданий по фамилии автора.

4. Создать абстрактный класс *Transport* с функциями, позволяющими вывести на экран информацию о транспортном средстве, а также определить грузоподъемность транспортного средства.

Создать производные классы: *Car* (марка, номер, скорость, грузоподъемность), *Motorbike* (марка, номер, скорость, грузоподъемность, наличие коляски (если коляска отсутствует, то грузоподъемность равна 0)), *Truck* (марка, номер, скорость, грузоподъемность, наличие прицепа (если есть прицеп, то грузоподъемность увеличивается в два раза)).

Создать базу (массив) из  $n$  машин, вывести полную информацию из базы на экран, а также организовать поиск машин, удовлетворяющих требованиям грузоподъемности.

5. Создать абстрактный класс *Persona* с функциями, позволяющими вывести на экран информацию о персоне, а также определить ее возраст (на момент текущей даты).

Создать производные классы: *Enrollee* (фамилия, дата рождения, факультет), *Student* (фамилия, дата рождения, факультет, курс), *Teacher* (фамилия, дата рождения, факультет, должность, стаж).

Создать базу (массив) из  $n$  персон, вывести полную информацию из базы на экран, а также организовать поиск персон, чей возраст попадает в заданный диапазон.

6. Создать абстрактный класс *Goods* с функциями, позволяющими вывести на экран информацию о товаре, а также определить, соответствует ли он сроку годности на текущую дату.

Создать производные классы: *Product* (название, цена, дата производства, срок годности), *Party* (название, цена, количество штук, дата производства, срок годности), *Kit* (название, цена, перечень продуктов).

Создать базу (массив) из  $n$  товаров, вывести полную информацию из базы на экран, а также организовать поиск просроченного товара (на момент текущей даты).

7. Создать абстрактный класс *Goods* с функциями, позволяющими вывести на экран информацию о товаре, а также определить, соответствует ли она исключому типу.

Создать производные классы: *Toy* (название, цена, производитель, материал, возраст, на который рассчитана), *Book* (название, автор, цена, издательство, возраст, на который рассчитана), *SportsEquipment* (название, цена, производитель, возраст, на который рассчитан).

Создать базу (массив) из  $n$  товаров, вывести полную информацию из базы на экран, а также организовать поиск товаров определенного типа.

8. Создать абстрактный класс *TelephoneDirectory* с функциями, позволяющими вывести на экран информацию о записях в телефонном справочнике, а также определить соответствие записи критерию поиска.

Создать производные классы: *Persona* (фамилия, адрес, номер телефона), *Organization* (название, адрес, телефон, факс, контактное лицо), *Friend* (фамилия, адрес, номер телефона, дата рождения).

Создать базу (массив) из  $n$  записей, вывести полную информацию из базы на экран, а также организовать поиск в базе по фамилии.

9. Создать абстрактный класс *Client* с функциями, позволяющими вывести на экран информацию о клиентах банка, а также определить соответствие клиента критерию поиска.

Создать производные классы: *Depositor* (фамилия, дата открытия вклада, размер вклада, процент по вкладу), *Credited* (фамилия, дата выдачи кредита, размер кредита, процент по кредиту, остаток долга), *Organization* (название, дата открытия счета, номер счета, сумма на счету).

Создать базу (массив) из  $n$  клиентов, вывести полную информацию из базы на экран, а также организовать поиск клиентов, начавших сотрудничать с банком с заданной даты.

10. Создать абстрактный класс *Software* с методами, позволяющими вывести на экран информацию о программном обеспечении, а также определить соответствие возможности использования (на момент текущей даты).

Создать производные классы: *FreeSoftware* (название, производитель), *SharewareSoftware* (название, производитель, дата установки, срок бесплатного использования), *ProprietarySoftware* (название, производитель, цена, дата установки, срок использования).

Создать базу (массив) из  $n$  видов программного обеспечения, вывести полную информацию из базы на экран, а также организовать поиск программного обеспечения, которое допустимо использовать на текущую дату.

11\*. Реализовать базу вакансий организаций, которая позволяет: добавлять и удалять информацию об организациях, добавлять и удалять информацию о вакансиях в организации; просматривать полную информацию о вакансиях или информацию о вакансиях в конкретной организации; осуществлять поиск вакансий по заданной специальности.

12\*. Реализовать каталог музыкальных компакт-дисков, который позволяет: добавлять и удалять диски; добавлять и удалять песни на диске; просматривать содержимое целого каталога и каждого диска в отдельности; осуществлять поиск всех песен заданного исполнителя по всему каталогу.

# Глава 16

## ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ РЕАЛИЗАЦИЯ СПИСКОВ

В данной главе мы рассмотрим структуры данных и алгоритмы, которые являются фундаментом современного программирования.

Термины «тип данных», «структура данных» и «абстрактный тип данных» звучат очень похоже, но имеют различный смысл.

В языках программирования *тип данных* определяет:

- внутреннее представление данных в памяти компьютера;
- объем оперативной памяти, выделяемый для размещения значения данного типа;
- множество значений, которые могут принимать величины этого типа;
- операции и стандартные функции, которые можно применять к величинам этого типа.

Например, в C++ целый тип *long*:

- имеет знаковый формат представления данных в памяти ЭВМ;
- в оперативной памяти занимает 4 байта;
- множество его значений лежит в диапазоне от  $-2\ 147\ 483\ 648$  до  $2\ 147\ 483\ 647$ ;
- к величинам этого типа могут применяться унарные операции ( $++, --, -, !$ ), бинарные арифметические операции ( $/, \%, *, +, -$ ), операции отношения ( $<, \leq, >, \geq, ==, !=$ ), логические операции ( $\&\&, ||$ ), операции присваивания ( $=, /=, \%=, *=, +=, -=$ ) и стандартные арифметические функции заголовочного файла *math.h*.

*Абстрактный тип данных* (АТД) — это математическая модель плюс различные операторы, определенные в рамках этой модели. Любой алгоритм может разрабатываться в терминах АТД. Но для реализации алгоритма на конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для этого используются *структуры данных*.

Более подробно мы рассмотрим АТД «список» и его реализацию с помощью структуры, называемой связанным списком.

## 16.1. Основные понятия

В математике **список** — это последовательность элементов  $a_1, a_2, \dots, a_n$  ( $n \geq 0$ ) одного типа, обладающая следующим характеристическим свойством: для любого  $i = 1, 2, \dots, n - 1$  элемент  $a_i$  предшествует  $a_{i+1}$ ; для любого  $i = 2, \dots, n$   $a_i$  следует за  $a_{i-1}$ , т.е. его элементы линейно упорядочены в соответствии с их позицией в списке.

Количество элементов  $n$  называется **длиной** списка. Если  $n > 0$ , то  $a_1$  называется **первым элементом** списка, а  $a_n$  — **последним элементом** списка. В случае  $n = 0$  имеем пустой список, который не содержит элементов.

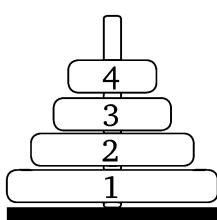
Однозначно определенного множества операторов для этой модели не существует. В каждом конкретном случае операции со списком будут зависеть от решаемой задачи. Однако можно выделить некоторые базовые операции: добавление элемента в список, удаление элемента из списка, проход по списку (например, для вывода элементов, для подсчета количества элементов с определенным свойством и т.д.).

Список может быть реализован в виде массива или с помощью указателей в виде связного списка.

Мы рассмотрим два вида связных списков: односвязный и двусвязный. Если каждый элемент списка содержит указатель на следующий элемент, то такой список называют **односвязным** или **однонаправленным**. Если каждый элемент списка содержит два указателя: один на следующий элемент в списке, второй — на предыдущий элемент, то такой список называется **двунаправленным** или **двусвязным**.

Далее подробно рассмотрим частные случаи списка (стек и очередь) и их односвязную реализацию, а также список общего вида в односвязной и двусвязной реализации.

## 16.2. Стек



**Стек** — это частный случай списка, все операции с которым (добавление, просмотр и выборка) выполняются с одного конца, называемого вершиной стека (головой — *head*). Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания *LIFO* (*last in — first out*, последним пришел — первым вышел). Стек проще всего представить себе в виде пирамиды, на которую надевают кольца.

Достать первое кольцо можно только после того, как будут сняты все верхние кольца.

Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах, поэтому очень важно разобраться в принципах работы со стеками.

Рассмотрим организацию стека с помощью однонаправленного списка, изображенного на рис. 16.1.

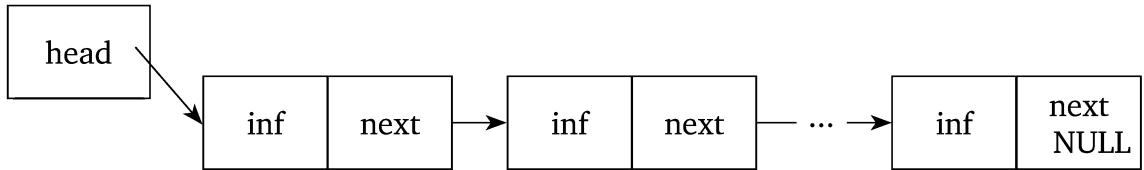


Рис. 16.1. Стек

Базовый элемент стека содержит:

- информационное поле *inf*, которое может быть любого типа, кроме файлового, и будет использоваться для хранения значений, например чисел, символов, строк;
- поле-указатель *next*, в котором хранится адрес следующего элемента в стеке, и которое будет использоваться для организации связи элементов стека.

#### Замечание

В общем случае базовый элемент стека может содержать несколько информационных полей.

Предложенный базовый элемент стека может быть описан следующим образом:

```

struct Element
{
    int inf;          // информационное поле целого типа
    Element *next;   // указатель на следующий элемент стека
    Element(int x, Element *p)
        : inf(x), next(p) // конструктор элемента стека
    {
    }
};

```

Как уже говорилось, доступ к элементам стека возможен только из одной точки — из вершины (головы) стека. Основные операции со стеком — это взять элемент с вершины и положить элемент на вершину. Можно еще добавить операцию просмотра верхнего элемента без его удаления из стека и проверку стека на пустоту. С учетом вышеизложенного представим теперь реализацию стека с помощью класса *Stack*:

```

class Stack
{
    struct Element
    {
        int inf;
        Element *next;
        Element(int x, Element *p) : inf(x), next(p) {}
    };
    Element *head; // указатель на вершину стека
public:
    Stack() : head(0) // конструктор стека

```

```

{}
bool Empty() // проверка стека на пустоту
{
    return head == 0;
}
int Pop() // взять элемент из стека
{
    if (Empty()) // если стек пуст, то ничего не делать
    {
        return 0;
    }
    // иначе запоминаем указатель на вершину стека
    Element *r = head;
    // запоминаем информацию из верхнего элемента
    int i = r->inf;
    // передвигаем указатель стека на следующий от вершины элемент
    head = r->next;
    delete r; // освобождаем память, на которую указывает r
    return i; // возвращаем значение
}
void Push(int data) // добавить элемент в стек
{
    head = new Element(data, head);
}
int Top() // просмотреть элемент на вершине стека
{
    if (Empty()) // если стек пуст, то возвращаем 0
    {
        return 0;
    } else // иначе возвращаем информацию из вершины стека
    {
        return head->inf;
    }
}
};


```

### Замечание

В операциях выборки и просмотра верхнего элемента мы осуществляли предварительную проверку стека на пустоту. Действительно, если стек пуст, то взять из него ничего нельзя. Однако функция должна возвратить какое-то значение целого типа, как определено при описании функции. В нашем примере мы возвращаем 0. Это возможно в том случае, если в стеке никогда не встретится элемент, равный 0. В общем случае используется механизм исключений, он будет рассмотрен ниже.

Рассмотрим базовые операции со стеком на примерах.

#### 1. Добавление элемента в стек.

```

void Push(int data)
{
    head = new Element(data, head);
}

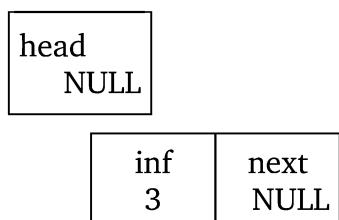
```

Теперь последовательно выполним следующие команды:

```
Stack t;      // 1
t.Push(3);   // 2
t.Push(8);   // 3
```

Команда 1 создает экземпляр класса *Stack* с именем *t*, при этом вершина стека пуста (поле *head* равно NULL).

Команда 2 приводит к выполнению оператора *head = new Element(3, head)*. Данный оператор состоит из двух частей:



- 1) выделение памяти под новый элемент списка (*new*), при этом в поле *data* нового элемента записывается значение 3, а в поле *next* записывается адрес области памяти, с которой связан указатель *head*;
- 2) переадресация указателя *head* на новый элемент.

Таким образом, после выполнения команды 2 стек содержит один элемент (рис. 16.2).

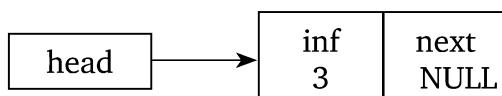
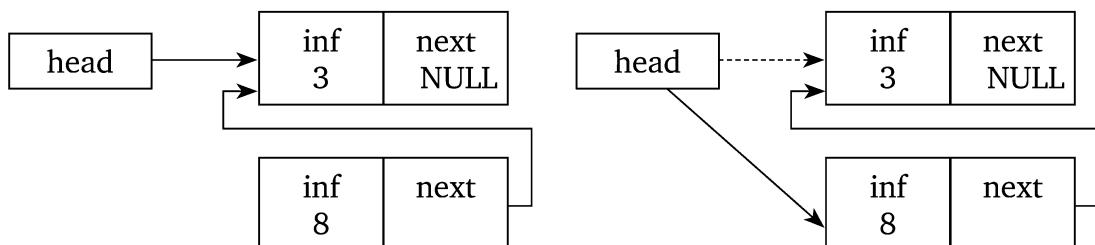


Рис. 16.2. Стек с одним элементом

Строка 3 приведет к выполнению оператора *head = new Element(8, head)*, который также последовательно выполнит два действия:



Таким образом, после выполнения команды 3 стек содержит два элемента (рис. 16.3).

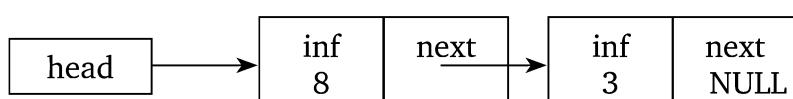


Рис. 16.3. Стек с двумя элементами

## 2. Извлечение элемента из стека.

```
int Pop()
{
    if (Empty()) // если стек пуст, то ничего не делать
    {
        return 0;
    }
}
```

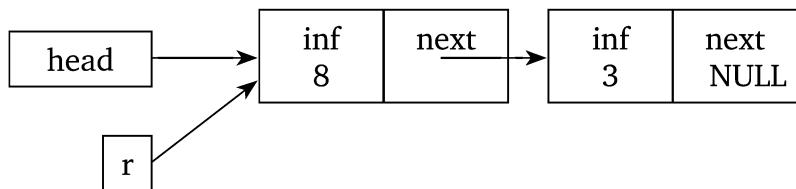
```

Element *r = head; // 1
int i = r->inf; // 2
head = r->next; // 3
delete r; // 4
return i; // 5
}

```

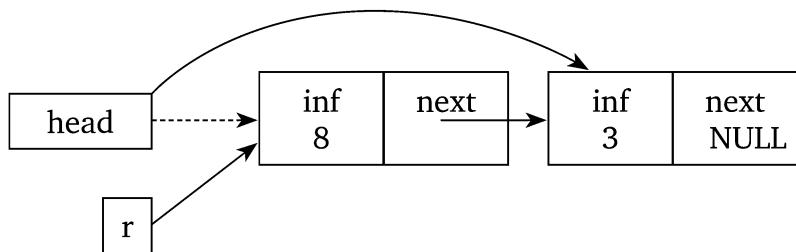
Проиллюстрируем работу функции *Pop*. Первоначально считаем, что структура стека соответствует рис. 16.3. Выполним операцию *t.Pop()*. Условие пустоты ложное (стек не пуст), поэтому последовательно выполняются следующие операторы.

Оператор 1: определяется указатель *r*, который устанавливается на верхний элемент стека.

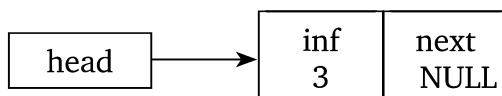


Оператор 2: в переменную *i* записывается 8 — значение информационного поля элемента, на который указывает *r*.

Оператор 3: указатель *head* переадресуется на элемент, следующий за *r* в стеке, таким образом, элемент со значением 8 исключается из стека, а верхним элементом становится элемент со значением 3.



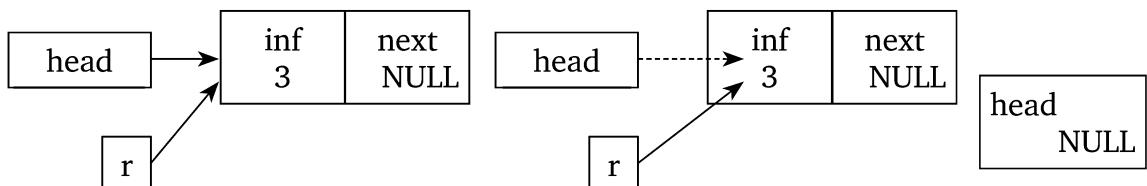
Оператор 4: освобождает область оперативной памяти, связанную с указателем *r*.



Оператор 5: функция возвращает значение переменной *i*, т.е. значение 8.

В результате выполнения операции *t.Pop()* стек состоит из одного элемента и соответствует рис. 16.2.

Повторное выполнение операции *t.Pop()* можно проиллюстрировать следующим образом.



Таким образом, стек окажется пустым.

Работу остальных членов-функций класса *Stack* рассмотрите самостоятельно.

Описание класса *Stack* оформим в виде отдельного файла с именем *stack.cpp*. В дальнейшем при решении практических задач будем подключать данный файл к проекту с помощью директивы `#include "stack.cpp"` после `using namespace std`. Обратите внимание на то, что сам файл *stack.cpp* должен находиться в той же папке, что и файл с кодом программы.

### 16.3. Решение практических задач с использованием стеков

1. Дан текстовый файл *input.txt*, в котором через пробел записаны натуральные числа. Переписать его содержимое в файл *output.txt* в обратном порядке.

```
#include <iostream>
using namespace std;
// подключение файла с реализацией класса Stack
#include "stack.cpp"
int main()
{
    Stack t;
    int i;
    ifstream in("input.txt");
    ofstream out("output.txt");
    while (in >> i) // пока не достигнут конец файла input.txt,
    {                   // читаем из него элементы
        t.Push(i);     // и помещаем их в стек
    }
    in.close();
    while (!t.Empty()) // пока стек не пуст
    {
        out << t.Pop() << " "; // извлекаем из него элементы
    }                   // и выводим в файл output.txt
    out.close();
    return 0;
}
```

Результат работы программы:

*input.txt*  
1 2 3 4 45 7 4 5 6 7 8 9 10 12

*output.txt*  
12 10 9 8 7 6 5 4 7 45 4 3 2 1

2. Данна последовательность натуральных чисел. Перед каждым элементом, равным *x*, вставить элемент, равный *y*. Входная последова-

тельность целых чисел и заданные числа  $x$  и  $y$  хранятся в файле *input.txt* (в первой строке файла через пробел записаны  $x$  и  $y$ , во второй — последовательность чисел), выходная последовательность целых чисел записывается в файл *output.txt*.

### Замечание

Идея решения заключается в следующем. Перепишем все числа из файла *input.txt* в стек. По свойству стека все числа там будут находиться в обратном порядке. Нам надо сохранить прямой порядок, а кроме того, вставить новые элементы. Поэтому введем второй стек и будем переписывать числа из первого во второй по следующему алгоритму. Берем число первого стека, переписываем во второй стек. Далее проверяем: если это число  $x$ , то во второй стек запишем число  $y$ . Таким образом, во втором стеке числа  $y$  будут вставлены после чисел  $x$ , но при извлечении из стека порядок поменяется на противоположный. Следовательно, остается только извлекать по одному элементу из второго стека и переписывать в выходной файл.

```
#include <fstream>
using namespace std;
// подключение файла с реализацией класса Stack
#include "stack.cpp"
int main()
{
    Stack t, t1; // инициализируем два стека
    int i, x, y;
    ifstream in("input.txt");
    ofstream out("output.txt");
    in >> x;
    in >> y;
    while (in >> i)
    {
        t.Push(i);
    }
    in.close();
    // пока стек t не пуст, переписываем из него числа
    // в стек t1, вставляя после каждого элемента
    // со значением x элемент со значением y
    while (!t.Empty())
    {
        i = t.Pop();
        t1.Push(i);
        if (i == x)
            t1.Push(y);
    }
    // переписываем содержимое стека t1 в выходной файл
    while (!t1.Empty())
    {
        out << t1.Pop() << " ";
    }
    out.close();
    return 0;
}
```

Результат работы программы:

*input.txt*  
4 0  
1 4 3 4 45 7 4 6 7 4

*output.txt*  
1 0 4 3 0 4 45 7 0 4 6 7 0 4

3. В файле находится текст, в котором имеются скобки (). Используя стек, проверить, соблюден ли баланс скобок в тексте.

### Замечание

Теперь стек должен обрабатывать символы. Поэтому в классе *Stack* тип информационного поля стека изменяется на *char*. Таким же образом будут изменены типы соответствующих параметров методов класса и типы возвращаемых значений методов класса.

```
#include <fstream>
using namespace std;
#include "stack.cpr"
// Функция, подсчитывающая баланс скобок. Возвращает:
// 0 - если баланс скобок соблюден;
// 1 - если имеется лишняя открывающая скобка;
// -1 - если имеется лишняя закрывающая скобка.
int Balans()
{
    // описываем стек, в который из всего текста будут
    // заноситься только открывающие скобки
    Stack t;
    char i;
    ifstream in("input.txt");
    while (in >> i)
    {
        switch (i)
        {
            // если считанный символ - (, то записываем его в стек
            case '(': t.Push(i);
            break;
            // если считанный символ ), то в стеке должна находиться
            // соответствующая ему (
            case ')':
                // если стек не пуст, то выбираем из него одну открывающую
                // скобку
                if (!t.Empty())
                {
                    t.Pop();
                }
                // иначе возвращаем -1 - код лишней закрывающей скобки
                else
                {
                    return -1;
                }
        }
    }
}
```

```

        }
    }
    in.close();
/* Если после того, как весь текст обработан, стек пустой, то баланс скобок в тексте соблюден – возвращаем код 0. В противном случае в стеке находятся открывающие скобки, которым не соответствуют закрывающие – возвращаем код 1. */
    if (t.Empty())
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
int main()
{
    ofstream out("output.txt");
    int i = Balans(); // вызываем функцию Balans
    // обрабатываем значение, возвращенное функцией Balans
    if (!i)
    {
        out << "ok";
    }
    else
    {
        if (i == 1)
        {
            out << "error: (";
        }
        else
        {
            out << "error: )";
        }
    }
    out.close();
    return 0;
}

```

Результаты работы программы:

<b>input.txt</b>	
(5-x*(4/x-(x+3)/(y-2)-y)-9	
(5-x)*(4/x-(x+3)/(y-2)-y)-9	
(5-x)*(4/x-(x+3)/(y-2)-y))-9	
7+9-x/2+4-b*3	

<b>output.txt</b>	
error: (	
ok	
error: )	
ok	

## 16.4. Применение исключений и шаблонов

В гл. 13 был рассмотрен механизм обработки исключительных ситуаций в C++. Данный механизм очень полезен при работе со стеками. Так,

извлечь элемент из стека или посмотреть значение верхнего элемента стека можно только тогда, когда стек не пуст. Чтобы обработать указанную ситуацию, мы будем использовать следующий класс:

```
#include <string>
using namespace std;
class StackException
{
private:
    string str;
public:
    StackException(string message) : str(message) {}
    string what()
    {
        return str;
    }
};
```

Данный класс оформим в отдельный файл *exception.cpp* и подключим его к файлу *stack.cpp*, в котором предложена реализация класса *Stack*.

В гл. 8 был рассмотрен механизм создания и использования шаблонов-функций. Аналогичным образом можно создавать классы-шаблоны. Рассмотрим подробно, как будет выглядеть класс *Stack* после добавления в него механизма обработки исключений и шаблонов:

```
#include "exception.cpp" // подключение класса StackException
// описание класса-шаблона, где Item - тип данных,
// переданный в класс
template <class Item>
class Stack //
{
    struct Element
    {
        Item inf;
        Element *next;
        Element(Item x, Element *p) : inf(x), next(p) {}
    };
    Element *head;
public:
    Stack() : head(0) {}
    bool Empty()
    {
        return head == 0;
    }
    Item Pop()
    {
        if (Empty()) // если стек пуст, то генерируем исключение
        {
            throw StackException("StackException: Pop – стек пуст");
        }
        else // иначе извлекаем элемент из стека
        {
            Element *r = head;
            Item i = r->inf;
```

```

        head = r->next;
        delete r;
        return i;
    }
}
void Push(Item data)
{
    head = new Element(data, head);
}
Item Top()
{
    if (Empty()) // если стек пуст, то генерируем исключение
    {
        throw StackException("StackException: Top – стек пуст");
    }
    else // иначе возвращаем значение верхнего элемента стека
    {
        return head->inf;
    }
}
};


```

#### Замечание

Далее при объектно-ориентированной реализации списков будем использовать классы-шаблоны.

---

## 16.5. Очередь

**Очередь** — это частный случай списка, добавление элементов в который выполняется в один конец (хвост — *tail*), а выборка и просмотр осуществляются с другого конца (головы — *head*). Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания *FIFO* (*first in — first out*, первым пришел — первым вышел). Очередь проще всего представить в виде узкой трубы, в один конец которой бросают мячи, а из другого конца они вылетают. Понятно, что мяч, который был брошен в трубу первым, первым и вылетит.



В программировании очереди применяются при математическом моделировании, диспетчеризации задач операционной системы, буферном вводе/выводе.

Рассмотрим организацию очереди с помощью структуры, изображенной на рис. 16.4. Отличие очереди от стека в том, что у очереди два указателя: первый (*head*) указывает на первый элемент очереди, второй (*tail*) — на последний элемент.

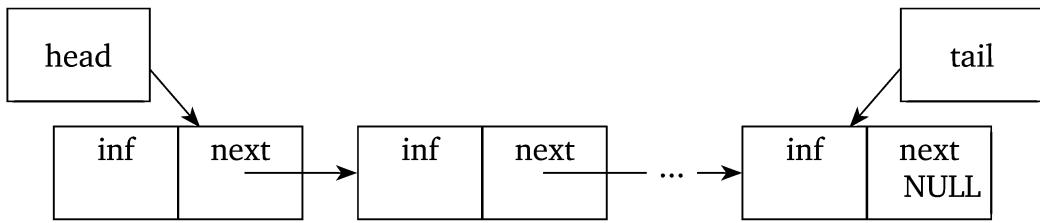


Рис. 16.4. Очередь

Базовый элемент очереди содержит:

- информационное поле *inf*, которое может быть любого типа, кроме файлового, и будет использоваться для хранения значений, например чисел, символов, строк;
- поле-указатель *next*, в котором хранится адрес следующего элемента очереди и которое будет использоваться для организации связи элементов.

#### Замечание

В общем случае базовый элемент очереди может содержать несколько информационных полей.

Предложенный базовый элемент очереди может быть описан следующим образом:

```

struct Element
{
    Item inf; // информационное поле типа Item
    Element *next; // указатель на следующий элемент очереди
    Element (Item x) : inf(x), next(0) // конструктор
    {
    }
};
  
```

Тогда указатели на первый и последний элементы очереди могут быть описаны следующим образом:

```
Element *head, *tail;
```

Как уже говорилось, доступ к элементам очереди возможен в двух точках: из одной точки — (головы) можно брать и просматривать элементы, во второй точке (хвост) можно добавлять элементы. С учетом вышеизложенного представим объектно-ориентированную реализацию очереди с помощью класса *Queue*:

```

#include "exception.cpp" // подключаем класс QueueException
// рассмотрим его позже
template <class Item> // класс-шаблон очередь
class Queue
{
    struct Element
    {
        Item inf;
        Element *next;
    };
  
```

```

Element(Item x) : inf(x), next(0) {}
};

Element *head, *tail; // указатели на начало и конец очереди
public:
Queue() : head(0), tail(0) {}
bool Empty() // возвращаем true, если очередь пуста, иначе false
{
    return head == 0;
}
Item Get()
{
    if (Empty()) // если очередь пуста, то генерируем исключение
    {
        throw QueueException("QueueException: get - queue empty");
    }
    else // иначе извлекаем элемент из головы очереди
    {
        Element *t = head;
        Item i = t->inf;
        head = t->next;
        if (head == NULL)
        {
            tail = NULL;
        }
        delete t;
        return i;
    }
}
void Put(Item data)
{
    // устанавливаем вспомогательный указатель
    // на последний элемент очереди
    Element *t = tail;
    // формируется новый элемент, на который будет
    // указывать tail
    tail = new Element(data);
    // если до этого очередь была пуста, то новый
    // элемент является и первым, и последним, поэтому
    // указатель head устанавливаем на этот элемент
    if (!head)
    {
        head = tail;
    }
    else
    {
        t->next = tail; // иначе новый узел помещаем в конец очереди
    }
}
};

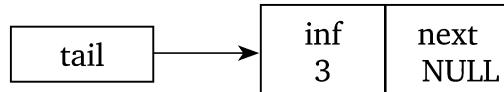
```

Более подробно рассмотрим базовые операции помещения элемента в очередь, предполагая, что у нас создан экземпляр класса *Queue<int>* *q*. Это значит, что выполнен конструктор, который создал пустую очередь с указателями: *head = NULL, tail = NULL*.

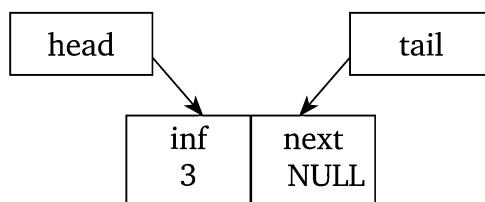
Первоначально добавим в очередь элемент 3, для чего выполним команду *q.Put(3)*.

Во вспомогательном указателе  $t$  (см. реализацию функции  $Put$ ) сохраняется значение указателя на последний элемент, т.е. `NULL`.

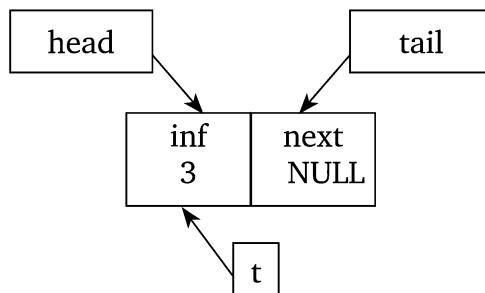
Затем происходит создание элемента, в информационную часть которого заносится значение 3, а в указатель  $next$  — значение `NULL`. После чего  $tail$  становится указателем на этот элемент.



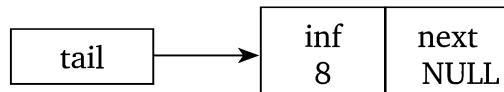
Далее производится проверка: пуста ли очередь (указывает ли  $head$  на какой-то элемент или его значение `NULL`)? В данном случае очередь пуста, поэтому  $head$  будет указывать туда же, куда и  $tail$ , т.е. на новый элемент со значением 3. Таким образом, получили очередь из одного элемента:



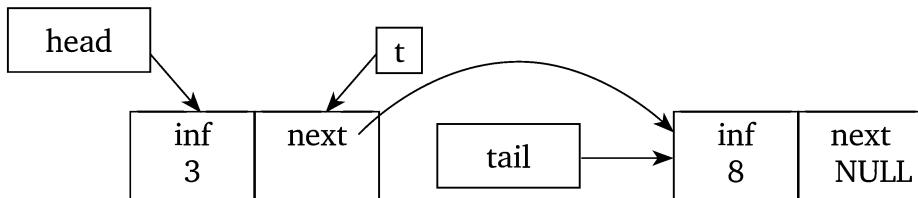
Теперь добавим в очередь элемент со значением 8, выполняя команду  $q.Put(8)$ . При этом создается вспомогательный указатель, который указывает на последний (в нашем случае единственный) элемент:



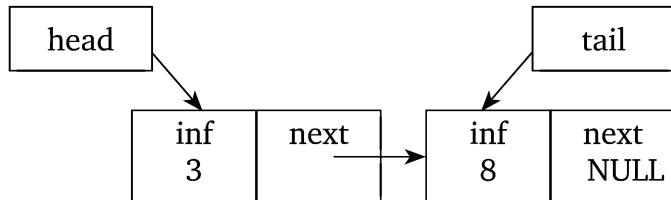
Затем происходит создание элемента, в информационную часть которого заносится значение 8, а в указатель  $next$  — значение `NULL`, после чего  $tail$  становится указателем на этот элемент:



Далее производится проверка: пуста ли очередь (указывает ли  $head$  на какой-то элемент или его значение `NULL`)? В данном случае очередь не пуста, поэтому следующим за элементом, на который указывает  $t$ , становится элемент, на который указывает  $tail$ .



В итоге получаем очередь из двух элементов, которые находятся в очереди в том же порядке, в каком их туда поместили.



Функция *Get*, которая извлекает верхний элемент из очереди, аналогична функции *Put* для стека. Отличием является следующий оператор, добавленный в функцию *Get*:

```
if (head == NULL)
{
    tail = NULL;
}
```

который говорит о том, что если после удаления элемента очередь становится пустой, то и указатель *tail* нужно «обнулить». Разберите данную функцию самостоятельно.

Функция *Empty* полностью совпадает с соответствующей функцией для стека.

#### Замечание

Класс *QueueException* помещен в отдельный файл *exception.cpp* и выглядит следующим образом:

```
#include "string"
using namespace std;
class QueueException
{
private:
    string str;

public:
    QueueException(string message) : str(message) {}
    string what()
    {
        return str;
    }
};
```

## 16.6. Решение практических задач с использованием очереди

1. Даны файлы *data1.txt* и *data2.txt*, компонентами которых являются натуральные числа. Переписать содержимое файла *data1.txt* в файл *data2.txt* и наоборот без использования вспомогательного файла.

```

#include <iostream>
#include <string>
using namespace std;
// подключаем файл с реализацией класса-шаблона очередь
#include "queue.cpp"
int main()
{
    Queue<int> t;
    int i;
    // описываем входной поток и связываем его с data1.txt
    ifstream in("data1.txt");
    while (in >> i) // пока не конец файла, считываем из него числа
    {
        t.Put(i); // кладем их в очередь
    }
    in.close(); // закрываем поток, связанный с файлом data1.txt
    // описываем входной поток и связываем его с data2.txt
    ifstream in1("data2.txt");
    // описываем выходной поток и связываем его с data1.txt
    ofstream out("data1.txt");
    // пока не конец файла data2.txt, считываем из него числа
    while (in1 >> i)
    {
        out << i << " "; // записываем в файл data1.txt
    }
    in1.close(); // закрываем поток, связанный с файлом data2.txt
    out.close(); // закрываем поток, связанный с файлом data1.txt
    // описываем выходной поток и связываем его с data2.txt
    ofstream out1("data2.txt");
    // пока очередь не пуста, выбираем из нее элементы
    // и записываем в data2.txt
    while (!t.Empty())
    {
        out1 << t.Get() << " ";
    }
    out1.close(); // закрываем поток, связанный с файлом data2.txt
    return 0;
}

```

Результат работы программы:

— до запуска программы:

**data1.txt**  
7 515 46 46

**data2.txt**  
2 5 8 12 45 6 6

— после запуска программы:

**data1.txt**  
2 5 8 12 45 6 6

**data2.txt**  
7 515 46 46

**2.** Данна последовательность натуральных чисел. Создать из них очередь и каждый ее элемент, равный  $x$ , заменить на элемент, равный  $y$ . Входная последовательность целых чисел и заданные числа  $x$  и  $y$  хранятся в файле *input.txt*, выходная последовательность целых чисел записывается в файл *output.txt*.

```
#include <fstream>
#include <string>
using namespace std;
// подключаем файл с реализацией класса-шаблона очередь
#include "queue.cpp"
int main()
{
    Queue<int> t;
    int i, x, y;
    ifstream in("input.txt");
    ofstream out("output.txt");
    in >> x;
    in >> y;
    // пока файл не пуст, считываем из него очередной элемент
    // если он равен  $x$ , то в очередь помещаем  $y$ ,
    // иначе исходное число
    while (in >> i)
    {
        if (i == x)
        {
            t.Put(y);
        }
        else
        {
            t.Put(i);
        }
    }
    in.close();
    // пока очередь не пуста,
    // извлекаем из нее элементы
    // и записываем в выходной файл
    while (!t.Empty())
    {
        out << t.Get() << " ";
    }
    out.close();
    return 0;
}
```

Результат работы программы:

**input.txt**  
7 0  
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

**output.txt**  
0 0 1 3 0 5 2 5 0 2 0 9 3 0 0

## 16.7. Однонаправленный список общего вида

Мы рассмотрели частные случаи списков (стек и очередь) и их односвязную реализацию. Заметим, что стек имеет одну «точку доступа», очередь — две «точки доступа». В общем случае список можно представить в виде линейной связанной структуры с произвольным количеством «точек доступа» (рис. 16.5), что позволяет определить следующие операции над списком: инициализация списка, добавление и удаление элемента из произвольного места списка, поиск элемента в списке по ключу, просмотр всех элементов без их извлечения списка.

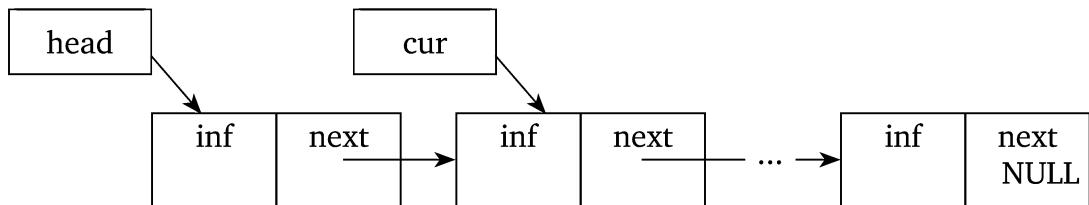


Рис. 16.5. Структура однонаправленного списка с указателями  
head на начало списка и cur — на произвольный элемент списка

Реализация однонаправленного списка зависит от решаемой задачи и предпочтений программиста. Рассмотрим объектно-ориентированную реализацию однонаправленного списка, представленного на рис. 16.5.

```
#include "exception.cpp"
template <class Item> class List
{
    struct Element
    {
        Item inf;
        Element *next;
        Element(Item x) : inf(x), next(0) {}
    };
    Element *head; // указатель на начало списка
    int size; // количество элементов в списке
    // возвращает указатель на элемент списка с номером index
    Element *Find(int index) {
        if ((index < 1) || (index > size)) // если индекс элемента списка
        { // находится вне диапазона, то
            return NULL; // возвращаем NULL
        }
        else // иначе
        {
            // устанавливаем указатель на начало списка
            Element *cur = head;
            for (int i = 1; i < index; i++) // и перемещаемся по списку
            { // на элемент с номером index
                cur = cur->next;
            }
            return cur; // возвращаем указатель на требуемый элемент
        }
    }
}
```

```

        }
    }
public:
    List()
        : head(0), size(0) // конструктор класса
    {}
    ~List() // деструктор класса
    {
        while (!Empty()) // пока список не пуст
        {
            Remove(1); // удаляем первый элемент списка
        }
    }
    bool Empty() // проверка пустоты списка
    {
        return head == 0;
    }
    int GetLength() // возвращает количество элементов в списке
    {
        return size;
    }
    // возвращает значение элемента списка по его номеру
    Item Get(int index)
    {
        if ((index < 1) || (index > size))
        {
            throw ListException("ListException: get - list error");
        }
        else
        {
            Element *r = Find(index);
            Item i = r->inf;
            return i;
        }
    }
    // осуществляет вставку элемента со значением data в позицию index
    void Insert(int index, Item data)
    {
        if ((index < 1) || (index > size + 1))
        {
            throw ListException("ListException: insert - list error");
        }
        else
        {
            // создаем новый элемент
            Element *newPtr = new Element(data);
            size = GetLength() + 1; // увеличиваем размерность списка
            if (index == 1) // если вставку производим в позицию 1,
            {
                // то см. рис. 16.6
                newPtr->next = head;
                head = newPtr;
            }
            else // иначе см. рис.16.7
            {
                Element *prev = Find(index - 1);

```

```

        newPtr->next = prev->next;
        prev->next = newPtr;
    }
}
// осуществляет удаление элемента из списка с номером index
void Remove(int index)
{
    if ((index < 1) || (index > size))
    {
        throw ListException("ListException: remove - list error");
    }
    else
    {
        Element *cur; // объявляем вспомогательный указатель
        --size;         // уменьшаем размерность списка
        if (index == 1) // если удаляем первый элемент,
        {               // то см. рис. 16.8
            cur = head;
            head = head->next;
        }
        else // иначе см. рис. 16.9
        {
            Element *prev = Find(index - 1);
            cur = prev->next;
            prev->next = cur->next;
        }
        cur->next = NULL;
        delete cur;
    }
}
// вывод элементов списка в глобальный поток out
void Print()
{
    for (Element *cur = head; cur != NULL; cur = cur->next)
    {
        out << cur->inf << ' ';
    }
    out << endl;
}
};

```

### Замечание

---

Класс *ListException* помещен в отдельный файл exception.cpp и выглядит следующим образом:

```

#include "string"
using namespace std;
class ListException
{
private:
    string str;
public:
    ListException(string message) : str(message) {}

```

```
string what()
{
    return str;
}
```

---

Более подробно основные операции с односторонними списками рассмотрим на примере членов-функций класса *List*.

### 1. Проход по списку (например, внутри члена-функции *Print*).

Для реализации этого действия необходим вспомогательный указатель, который вначале устанавливается на первый элемент списка, а потом «пробегает» весь список, указывая поочередно на каждый элемент.

Чтобы установить указатель на первый элемент списка, выполняем команду:

```
Element *cur = head;
```

Для вывода содержимого текущего узла в глобальный поток *out* используется оператор:

```
out << cur->inf;
```

Для перехода на следующий узел используется команда:

```
cur = cur->next;
```

И заканчивается проход по списку тогда, когда указатель *cur* примет значение *NULL*.

В результате получаем цикл:

```
for (Element *cur = head; cur != NULL; cur = cur->next)
{
    cout << cur->inf << ' ';
```

#### Замечание

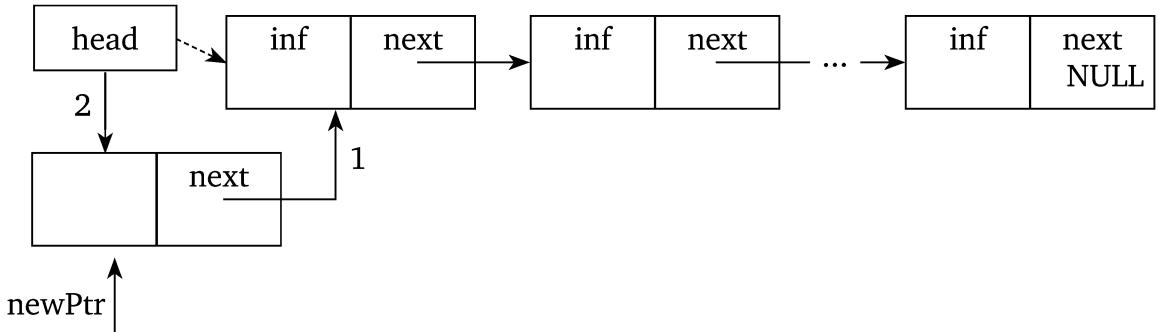
Аналогичный прием используется в функции *Find*, которая возвращает указатель на элемент списка с номером *index*. Отличие только в том, что мы завершаем перемещение по списку тогда, когда найдем элемент с заданным номером.

### 2. Вставка нового элемента в список в заданную позицию (на примере члена-функции *Insert*).

Существует два варианта вставки элемента в список.

Вариант первый — это вставка элемента в начало списка (рис. 16.6). В этом случае необходимо последовательно выполнить две команды:

```
newPtr->next = head; // 1
head = newPtr;          // 2
```



**Рис. 16.6. Вставка элемента в начало списка**

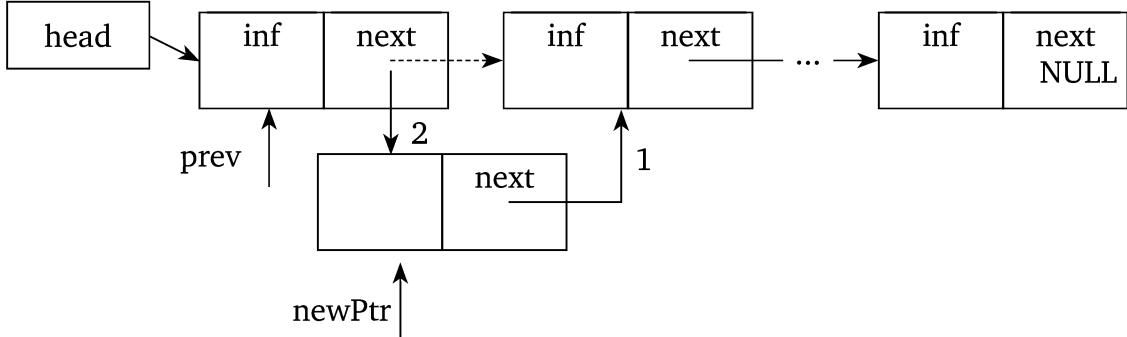
Заметим, что вставка в пустой список — это вставка элемента в начало списка, у которого *head* = NULL.

Второй вариант — вставка элемента в середину списка (рис. 16.7). В этом случае необходимо знать указатель на предшествующий элемент. Узнать его можно с помощью команды:

```
Element *prev = Find(index-1);
```

Следующие команды позволяют нам вставить новый элемент в заданную позицию:

```
newPtr->next = prev->next; // 1
prev->next = newPtr; // 2
```



**Рис. 16.7. Вставка элемента в середину списка**

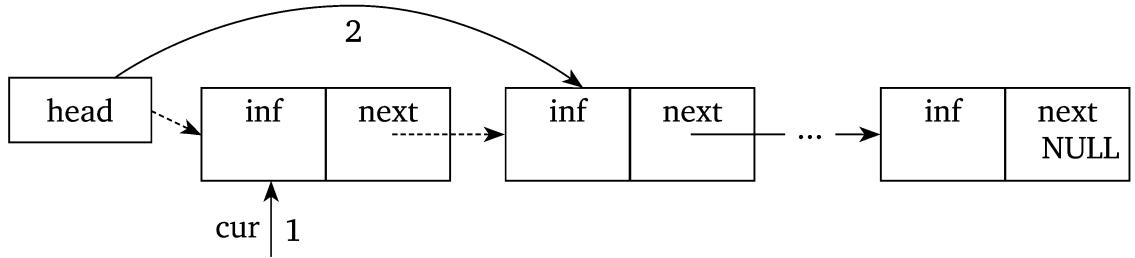
Заметим, что вставка элемента в конец списка пройдет корректно, так как указатель *prev* определен и указатель *prev->next* равен NULL.

**3. Удаление элемента из списка** (на примере члена-функции *Remove*).

Так же, как и при вставке элемента в список, существует два варианта.

Первый вариант — удаление первого элемента из списка (рис. 16.8). В этом случае нам необходимо последовательно выполнить две команды:

```
cur = head; // 1
head = head->next; // 2
```



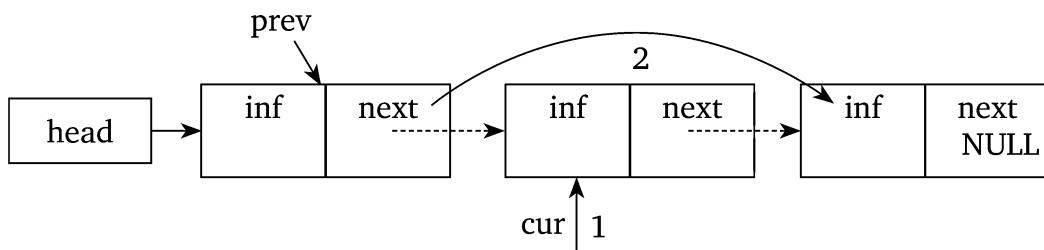
**Рис. 16.8. Удаление первого элемента из списка**

Второй вариант — вставка элемента в середину списка (рис. 16.9). В этом случае нам необходимо знать указатель на предшествующий элемент. Это можно сделать с помощью команды

```
Element *prev = Find(index-1);
```

Следующие команды позволяют нам удалить требуемый элемент:

```
cur = prev->next;      // 1
prev->next = cur->next; // 2
```



**Рис. 16.9. Удаление элемента из середины списка**

Заметим, что удаление элемента из конца списка пройдет корректно, так как указатель *prev* определен и указатель *prev->next* равен NULL.

#### Замечание

При работе со стеком и очередью нам не требовался деструктор, так как обработка стека и очереди подразумевает извлечение из них элементов, в результате чего в конце работы указанные виды списков остаются пустыми. При работе со списками общего вида элементы из них могут и не извлекаться. Поэтому требуется корректно освободить память, выделенную под список, что и делает деструктор.

## 16.8. Решение практических задач с использованием односторонних списков

- Дана последовательность натуральных чисел. Создать из них список и после каждого элемента, равного *x*, вставить элемент, равный *y*. Входная последовательность целых чисел и заданные числа *x* и *y* хра-

нятся в файле *input.txt*, выходная последовательность целых чисел записывается в файл *output.txt*.

```
#include <fstream>
#include <string>
using namespace std;
// подключаем файл с реализацией класса-шаблона список
#include "list.cpp"
// подключаем глобальные потоки
ifstream in("input.txt");
ofstream out("output.txt");
int main()
{
    List<int> l;
    int value, x, y;
    in >> x;
    in >> y;
    // пока файл не пуст, считываем из него очередной элемент
    // и помещаем в список; позиция, в которую вставляем новый элемент,
    // будет на 1 больше, чем количество элементов в списке
    while (in >> value)
    {
        l.Insert(l.GetLength() + 1, value);
    }
    in.close();
    out << "Исходный список: ";
    l.Print();
    // просматриваем список поэлементно
    for (int i = 1; i <= l.GetLength(); i++) {
        if (l.Get(i) == x) // если значение элемента
                           // в i-й позиции равно x
        {
            l.Insert(i + 1, y); // вставляем элемент у в эту позицию
            i++;
        }
    }
    out << "Измененный список: ";
    l.Print();
    l~List(); // вызываем деструктор
    out.close();
    return 0;
}
```

Результат работы программы:

*input.txt*

```
7 0
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7
```

*output.txt*

```
Исходный список: 7 7 1 3 7 5 2 5 7 2 7 9 3 7 7
Измененный список: 7 0 7 0 1 3 7 0 5 2 5 7 0 2 7 0 9 3 7 0 7 0
```

2. Данна последовательность натуральных чисел. Создать из них очередь и каждый ее элемент, равный  $x$ , заменить элементом, равным  $y$ .

Входная последовательность целых чисел и заданные числа  $x$  и  $y$  хранятся в файле *input.txt*, выходная последовательность целых чисел записывается в файл *output.txt*.

Решение данной задачи можно свести к применению двух членовых функций, *Insert* и *Remove*. Так, в функции *main* мы перебираем все элементы списка, если встречаем элемент, равный  $x$ , то удаляем его, а на его место вставляем элемент, равный  $y$ . Этот алгоритм можно реализовать следующим способом:

```
for (int i = 1; i <= l.GetLength(); i++)
{
    if (l.Get(i) == x)
    {
        l.Remove(i);
        l.Insert(i,y);
    }
}
```

Однако для решения данной задачи более рационально добавить новую член-функцию *Change* в класс *List*. В функции *Change* мы просматриваем все элементы списка. Если значение очередного элемента совпадает со значением  $x$ , то заменяем значение этого элемента на  $y$ . Этот алгоритм можно реализовать следующим способом:

```
void Change(Item x, Item y)
{
    // устанавливаем указатель на начало списка
    Element *cur = head;
    // перебираем элементы списка по очереди
    for (Element *cur = head; cur != NULL; cur = cur->next)
    {
        if (cur->inf == x) // если элемент равен x
        {
            cur->inf = y; // заменяем его на y
        }
    }
}
```

Второй способ решения задачи соответствует идеологии ООП, а добавление новой член-функции в класс *List* существенно расширяет его функциональные возможности.

Теперь полное решение данной задачи выглядит следующим образом:

```
#include <fstream>
#include <string>
using namespace std;
// не забудьте добавить функцию Change в класс List
#include "list.cpp"
ifstream in("input.txt");
ofstream out("output.txt");
int main()
{
    List<int> l;
```

```

int value, x, y;
in >> x;
in >> y;
while (in >> value)
{
    l.Insert(l.GetLength() + 1, value);
}
in.close();
out << "Исходный список: ";
l.Print();
l.Change(x, y); // вызов член-функции Change
out << "Измененный список: ";
l.Print();
l~List();
out.close();
return 0;
}

```

Результат работы программы:

*input.txt*

```

7 0
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

```

*output.txt*

```

Исходный список: 7 7 1 3 7 5 2 5 7 2 7 9 3 7 7
Измененный список: 0 0 1 3 0 5 2 5 0 2 0 9 3 0 0

```

## 16.9. Двунаправленный список

До сих пор мы рассматривали односторонние списки. В таких списках можно передвигаться лишь в одном направлении, а чтобы, например, удалить элемент, необходимо иметь указатель на предыдущий элемент. Однако иногда требуется одинаково эффективно передвигаться по списку в обоих направлениях. В этом случае используют двунаправленные списки, когда в каждом узле имеются два указателя: на следующий и на предыдущий элемент (рис. 16.10).

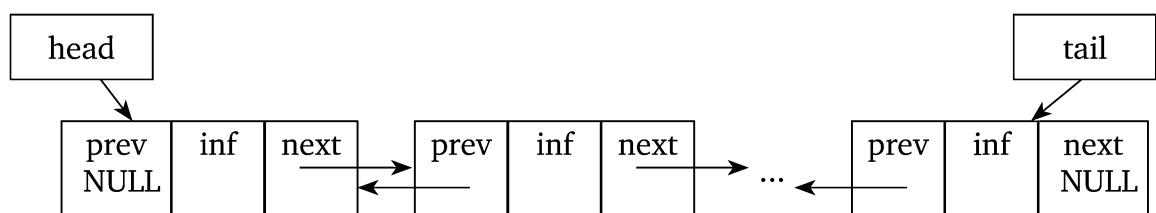


Рис. 16.10. Структура двунаправленного списка

Базовый элемент двунаправленного списка содержит:

- информационное поле *inf*, которое может быть любого типа, кроме файлового, и будет использоваться для хранения значений, например чисел, строк, записей;

— поля-указатели *next* и *prev*, в которых хранятся адреса следующего и предыдущего элементов двунаправленного списка соответственно, и которые будут использоваться для организации связи элементов.

### Замечание

---

В общем случае базовый элемент списка может содержать несколько информационных полей.

---

Предложенный базовый элемент двунаправленного списка может быть описан следующим образом:

```
struct Element
{
    Item inf;           // информационное поле типа Item
    Element *next;     // указатель на следующий элемент списка
    Element *prev;     // указатель на предыдущий элемент списка
    Element (Item x): inf(x), next(0), prev(0)
    {
    }
}
```

Поскольку в каждом узле такого списка задействовано два указателя вместо одного, механизм операций вставки и удаления будет несколько усложнен по сравнению с односвязными списками. Зато можно удалять элемент, на который есть указатель, и вставлять элементы до и после заданного, не задействуя дополнительные указатели.

Приведем объектно-ориентированную реализацию двунаправленного списка, а затем подробно рассмотрим базовые операции с данным видом списка.

```
#include "exception.cpp"
template <class Item> class DoubleLinkedList
{
    struct Element
    {
        Item inf;
        Element *next;
        Element *prev;
        Element(Item x) : inf(x), next(0), prev(0) {}
    };
    Element *head;
    Element *tail;
    int size;
    // Возвращает указатель на элемент списка с номером index
    Element *Find(int index)
    {
        if ((index < 1) || (index > size))
        {
            return NULL;
        }
        else
        {
```

```

Element *cur = head;
for (int i = 1; i < index; i++)
{
    cur = cur->next;
}
return cur;
}
}
public:
DoubleLinkedList() : head(0), tail(0), size(0) // конструктор
{}
~DoubleLinkedList() // деструктор
{
    while (!Empty())
    {
        Remove(1);
    }
}
bool Empty() // проверяет список на пустоту
{
    return head == 0;
}
int GetLength() // возвращает количество элементов в списке
{
    return size;
}
// возвращает значение элемента списка по его номеру
Item Get(int index)
{
    if ((index < 1) || (index > size))
    {
        throw DoubleListException(
            "Exception: get - double-linked list error");
    }
    else
    {
        Element *r = Find(index);
        Item i = r->inf;
        return i;
    }
}
// осуществляет вставку элемента со значением data
// слева от элемента с позицией index
void InsertLeft(int index, Item data)
{
    if ((index < 1) || (index > size + 1))
    {
        throw DoubleListException(
            "Exception: insert - double-linked list error");
    }
    else
    {
        Element *newPtr = new Element(data);
        size = GetLength() + 1; // увеличиваем размерность списка

```

```

// устанавливаем указатель на элемент списка
// с заданным номером
Element *cur = Find(index);
// если этот указатель NULL, то список был пуст, поэтому
// новый элемент будет и первым, и последним
if (cur == NULL) // см. рис. 16.11.
{
    head = newPtr;
    tail = newPtr;
} else
// иначе производим вставку в непустой список, при этом
// есть два случая: 1 - частный случай (вставка перед
// первым элементом списка), 2 - общий случай
{
    newPtr->next = cur;
    newPtr->prev = cur->prev;
    cur->prev = newPtr;
    if (cur == head) {
        head = newPtr; // случай 1
    }
    else
    {
        newPtr->prev->next = newPtr; // случай 2
    }
}
}

// осуществляет вставку элемента со значением data
// слева от элемента с позицией index
void InsertRight(int index, Item data)
{
    if ((index < 1 &&head != NULL) || (index > size + 1))
    {
        throw DoubleListException(
            "Exception: insert - double-linked list error");
    }
    else
    {
        Element *newPtr = new Element(data);
        size = GetLength() + 1; // увеличиваем размерность списка
        // устанавливаем указатель на элемент списка
        // с заданным номером
        Element *cur = Find(index);
        // если этот указатель NULL, то список был пуст, поэтому
        // новый элемент будет и первым, и последним
        if (cur == NULL) {
            head = newPtr;
            tail = newPtr;
        }
        // иначе производим вставку в непустой список, при этом
        // есть два случая: 1 - вставка после последнего элемента
        // списка, 2 - вставка в середину списка
        else
        {
            newPtr->next = cur->next;

```

```

    newPtr->prev = cur;
    cur->next = newPtr;
    if (cur == tail)
    {
        tail = newPtr; // случай 1
    }
    else
    {
        newPtr->next->prev = newPtr; // случай 2
    }
}
}

// осуществляем удаление элемента с номером index из списка
// выделяем четыре случая: 1 - после удаления список становится
// пустым, 2 - удаляем первый элемент, 3 - удаляем последний
// элемент, 4 - общий случай
void Remove(int index)
{
    if ((index < 1) || (index > size))
    {
        throw DoubleListException(
            "Exception: remove - double-linked list error");
    }
    else
    {
        // устанавливаем указатель на заданный элемент
        Element *cur = Find(index);
        --size;           // уменьшаем размерность списка
        if (size == 0) // случай 1
        {
            head = NULL;
            tail = NULL;
        }
        else if (cur == head) // случай 2
        {
            head = head->next;
            head->prev = NULL;
        }
        else if (cur == tail) // случай 3
        {
            tail = tail->prev;
            tail->next = NULL;
        }
        else // общий случай, см.рис. 16.14
        {
            cur->prev->next = cur->next;
            cur->next->prev = cur->prev;
        }
        cur->next = NULL;
        cur->prev = NULL;
        delete cur;
    }
}

```

```
// вывод элементов списка в глобальный поток out в прямом порядке
void PrintLeftToRight()
{
    for (Element *cur = head; cur != NULL; cur = cur->next)
    {
        out << cur->inf << ' ';
    }
    out << endl;
}
// вывод элементов списка в глобальный поток out в обратном порядке
void PrintRightToLeft()
{
    for (Element *cur = tail; cur != NULL; cur = cur->prev)
    {
        out << cur->inf << ' ';
    }
    out << endl;
}
};
```

---

### Замечание

Класс *DoubleListException* помещен в отдельный файл *exception.cpp* и выглядит следующим образом:

```
#include "string"
using namespace std;
class DoubleListException
{
private:
    string str;
public:
    DoubleListException(string message) : str(message) {}
    string what()
    {
        return str;
    }
};
```

---

Более подробно основные операции с двунаправленными списками рассмотрим на примере членов-функций класса *DoubleLinkedList*.

1. **Проход по списку в обратном порядке** (на примере члена-функции *PrintRightToLeft*).

Для реализации этого действия необходим вспомогательный указатель, который вначале устанавливается на последний элемент списка, а потом «пробегает» весь список в обратном порядке по указателям *prev*.

Чтобы установить указатель на последний элемент списка, выполняем команду

```
Element *cur = tail;
```

Для вывода содержимого текущего узла в глобальный поток *out* используется оператор

```
out << cur->inf;
```

Для перехода на следующий узел используется команда:

```
cur = cur->prev;
```

И заканчивается проход по списку тогда, когда указатель *cur* примет значение NULL.

В результате получаем цикл

```
for (Element *cur = tail; cur != NULL; cur = cur->prev)
{
    out << cur->inf << ' ';
}
```

#### Замечание

Проход по списку в прямом порядке используется в функции *PrintLeftToRight*. Рассмотрите данную функцию самостоятельно.

## 2. Вставка нового элемента в список слева от заданной позиции (на примере член-функции *InsertLeft*).

При вставке нового элемента в двунаправленный список слева от заданной позиции нужно рассмотреть несколько возможных ситуаций.

Если первоначально список пустой, то новый элемент станет одновременно первым и последним. Для этого необходимо установить указатели *head* и *tail* на новый элемент:

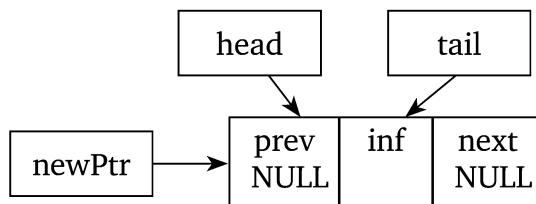


Рис. 16.11. Добавление элемента в пустой список

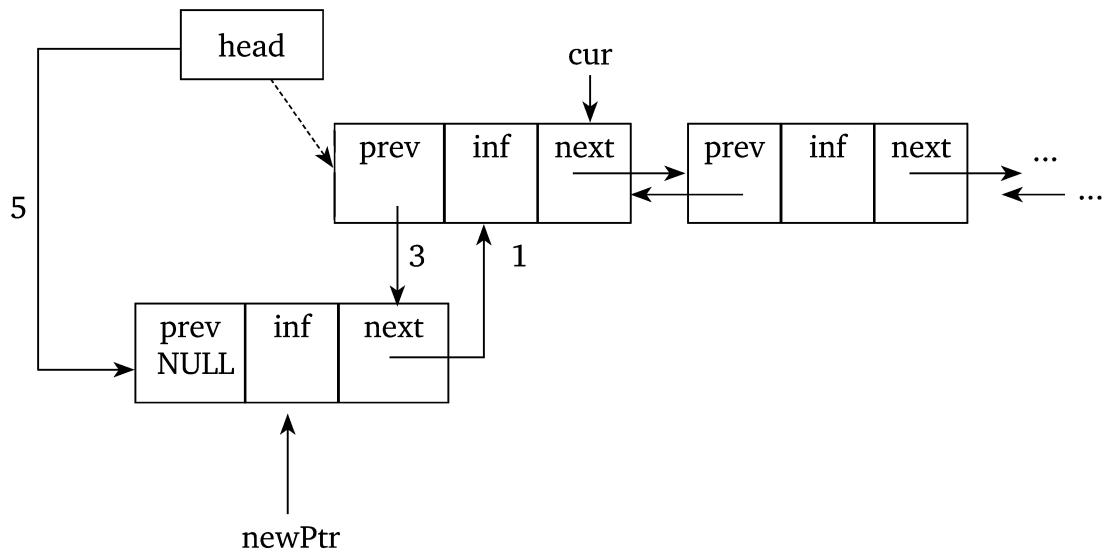
Если список не пустой, то возможны два случая: частный — вставка перед первым элементом в списке (рис. 16.12), и общий случай — вставка в произвольное место списка (рис. 16.13). Рассмотрим фрагмент функции *InsertLeft* более подробно.

```
newPtr->next = cur;           // 1
newPtr->prev = cur->prev;      // 2
cur->prev = newPtr;            // 3
if (cur == head)               // 4
{
    head = newPtr;             // 5
}
```

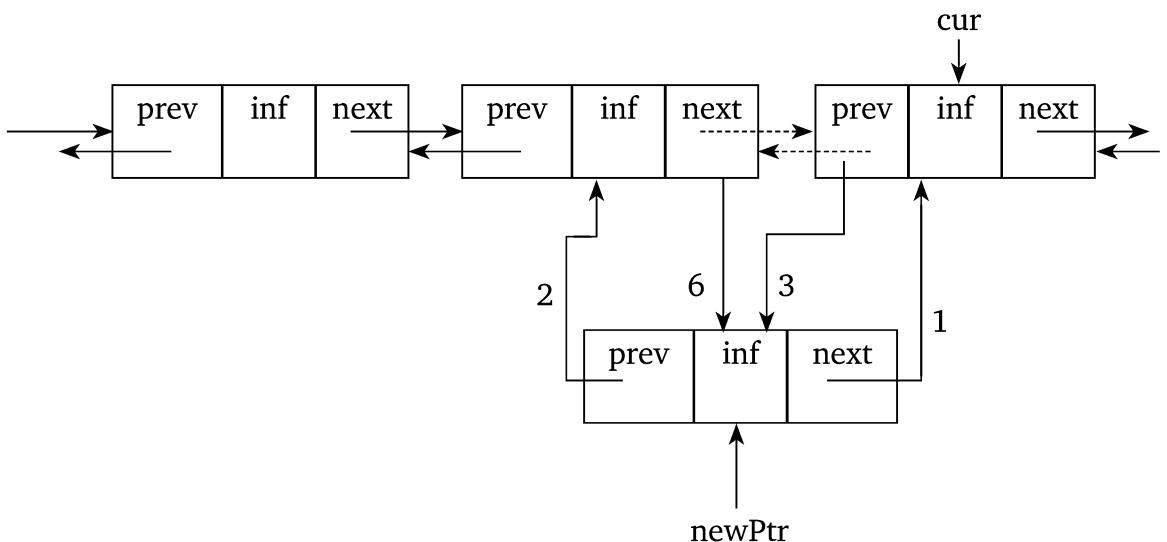
```

else
{
    newPtr->prev->next = newPtr; // 6
}

```



**Рис. 16.12. Частный случай вставки нового элемента в начало двунаправленного списка**



**Рис. 16.13. Общий случай вставки нового элемента в двунаправленный список слева от заданной позиции**

Как видно из рисунков, вставка элемента в список требует аккуратной работы с указателями. В противном случае будет нарушена целостность списка, а также последовательность расположения элементов в списке.

#### Замечание

Аналогичным образом разберите функцию *InsetRigth*.

### 3. Удаление элемента с заданным номером из двунаправленного списка (на примере функции *Remove*).

При удалении элемента с заданным номером из двунаправленного списка необходимо рассмотреть четыре случая. Продемонстрируем работу фрагментов функции *Remove*, отвечающих за обработку каждого из возможных случаев.

Случай 1 (частный) — после удаления список становится пустым:

```
// если после удаления элемента список окажется пустым, то
// указатели на начало и конец списка «обнулим»
if (size == 0)
{
    head = NULL ;
    tail = NULL;
}
```

Случай 2 (частный) — удаляем первый элемент в списке:

```
// если удаляем первый элемент из списка, то указатель
// head перемещаем на следующий элемент и полагаем,
// что у этого элемента предыдущего нет
if (cur == head)
{
    head = head->next;
    head->prev = NULL;
}
```

Случай 3 (частный) — удаляем последний элемент в списке:

```
// если удаляемый элемент последний, то указатель tail
// перемещаем на предыдущий элемент и полагаем,
// что у этого элемента следующего нет
if (cur == tail)
{
    tail = tail->prev;
    tail->next = NULL;
}
```

Случай 4 (общий) — удаляем произвольный элемент в списке:

```
cur->prev->next = cur->next; // 1
cur->next->prev = cur->prev; // 2
```

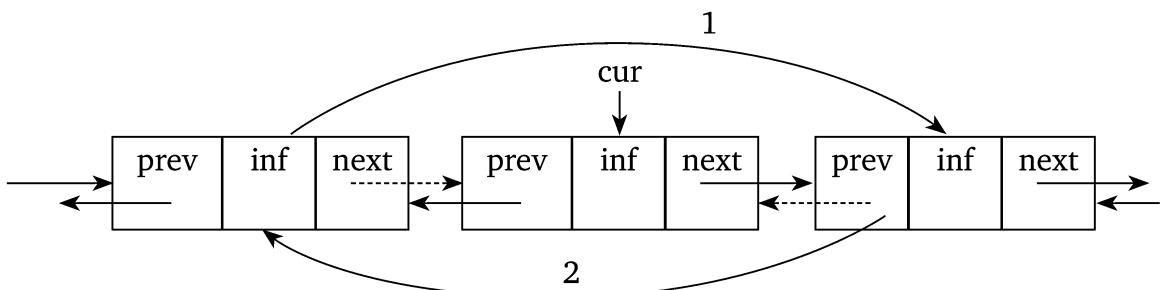


Рис. 16.14. Общий случай удаления элемента с заданным номером из двунаправленного списка

Затем, независимо от того, какой именно случай нам встретился, выполняется последовательность команд:

```
// для удаляемого элемента «обнуляем» указатели  
// на следующий и предыдущий элементы, чтобы  
// корректно исключить его из списка  
cur->next = NULL;  
cur->prev = NULL;  
// освобождаем память, связанную с указателем cur  
delete cur;
```

## 16.10. Решение практических задач с использованием дву направленных списков

1. Данна последовательность натуральных чисел. Создать из них список, после чего удалить из списка все элементы, равные  $x$ . Число  $x$  и входная последовательность целых чисел хранятся в файле *input.txt*, выходная последовательность целых чисел записывается в файл *output.txt*.

```
#include <iostream>  
#include <string>  
using namespace std;  
// подключаем файл, содержащий класс-шаблон DoubleLinkedList  
#include "double_linked_list.cpp"  
// определяем глобальные потоки  
ifstream in("input.txt");  
ofstream out("output.txt");  
int main()  
{  
    DoubleLinkedList<int> l;  
    int value, x;  
    in >> x;  
    // пока файл не пуст, считываем из него очередной элемент и  
    // помещаем в список  
    while (in >> value)  
    {  
        l.InsertRight(l.GetLength(), value);  
    }  
    in.close();  
    out << "Исходный список: ";  
    l.PrintLeftToRight();  
    // просматриваем список поэлементно  
    for (int i = 1; i <= l.GetLength());  
    {  
        if (l.Get(i) == x) // если значение элемента  
                           // в i-й позиции равно x  
        {  
            l.Remove(i); // то удаляем данный элемент из списка  
        }  
        else
```

```

    {
        i++; // иначе переходим к следующему элементу списка
    }
}
out << "Измененный список: ";
l.PrintLeftToRight();
out.close();
l.~DoubleLinkedList(); // вызываем деструктор
return 0;
}

```

Результат работы программы:

**input.txt**

```

7
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

```

**output.txt**

```

Исходный список: 7 7 1 3 7 5 2 5 7 2 7 9 3 7 7
Измененный список: 1 3 5 2 5 2 9 3

```

2. Дан файл, компонентами которого являются целые числа. Создать на основе данных файла упорядоченную по убыванию структуру данных без использования какого-либо алгоритма сортировки.

```

#include <iostream>
#include <string>
using namespace std;
#include "double_linked_list.cpp"
ifstream in("input.txt");
ofstream out("output.txt");
int main()
{
    DoubleLinkedList<int> l;
    int value, i, k = 1;
    in >> value; // считываем первый элемент из файла
    l.InsertRight(l.GetLength(), value); // помещаем его в список
    // последовательно считываем элементы из файла
    while (in >> value)
    {
        // пропускаем все элементы списка, большие value
        for (i = 1; ((i < k) && (l.Get(i) > value)); i++);
        // если дошли до конца списка и все числа больше value
        if (i == k && l.Get(i) > value)
        {
            // то вставляем value в конец списка
            l.InsertRight(i, value);
        }
        // иначе нашли элемент в списке, меньший value,
        // и вставляем value слева от этого элемента
        else
        {
            l.InsertLeft(i, value);
        }
        k++; // увеличиваем количество элементов в списке
    }
}

```

```

    }
    in.close();
    l.PrintLeftToRigh();
    out.close();
    l.~DoubleLinkedList();
    return 0;
}

```

Результат работы программы:

**input.txt**  
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

**output.txt**  
9 7 7 7 7 7 5 5 3 3 2 2 1

## Упражнения

I. Каждую задачу данного параграфа нужно выполнить в четырех вариантах: со стеком, очередью, списком общего вида в односвязной и двусвязной реализации. При необходимости функциональные возможности классов-шаблонов *List* и *DoubleLinkedList* можно расширить. Ввод-вывод данных файловый.

1. Создать список из целых чисел. Подсчитать количество отрицательных элементов, создав из них новый список.
2. Создать список из целых чисел. Подсчитать сумму положительных элементов. Создать из положительных элементов новый список.
3. Создать список из целых чисел. Найти среднее арифметическое значение четных элементов списка. Создать из четных элементов новый список.
4. Создать список из слов. Подсчитать, сколько слов начинается на данную букву. Создать из них новый список, удалив их из исходного списка.
5. Создать список из целых чисел. После каждого элемента, равного  $x$ , вставить элемент, равный  $y$ .
6. Создать список из целых чисел. Исключить из списка все элементы, равные  $x$ .
7. Создать список из целых чисел. Заменить каждую последовательность повторяющихся элементов на один элемент.
8. Создать список из целых чисел. Удвоить вхождение каждого четного элемента в нем.
9. Создать список из целых чисел. Найти минимальный элемент и удалить из списка все элементы, равные минимальному.
10. Создать список из целых чисел. Найти максимальный элемент и удалить из списка все элементы, равные максимальному.
11. Создать список из целых чисел. Поменять в списке местами максимальный и минимальный элементы.
12. Создать список из слов. Подсчитать количество слов, совпадающих с последним словом. Удалить все такие слова из списка, оставив одно последнее.
13. Создать список из слов, в который все слова исходного текста входят только один раз.
14. Создать список из целых чисел. Создать новый список, записав в него вначале все четные, а затем все нечетные числа из исходного списка.
15. Создать список из целых чисел. Создать новый список, записав в него вначале все положительные числа, а затем все отрицательные числа из исходного списка.

16. Создать список из чисел. Подсчитать количество пар соседних элементов, которые совпадают между собой. Оставить по одному из таких элементов, т.е. исключить все повторяющиеся, идущие подряд элементы.

17. Создать список из чисел. Создать новый список, записав в него сначала все отрицательные элементы из исходного списка, затем все положительные.

18. Создать список из целых чисел. Удалить лишние элементы в списке так, чтобы в результирующем списке каждый элемент был не меньше среднего арифметического всех элементов, следующих за ним.

19. Создать список из слов. Продублировать все однобуквенные слова в списке.

20. Создать список из слов. Перестроить элементы списка в обратном порядке.

II\*. Пусть дано математическое выражение, в котором используются лексемы (синтаксически неделимые единицы):

- 1) целые и действительные числа;
- 2) математические операции: +, -, \*, /;
- 3) круглые скобки;
- 4) однобуквенные переменные.

Для программного подсчета значения математического выражения необходимо:

- 1) разбить данное математическое выражение на лексемы;
- 2) проверить корректность математической записи;
- 3) записать выражение в виде обратной польской записи;
- 4) по записи подсчитать значение выражения (если в выражении встречаются переменная, то ее значение запрашивается с клавиатуры только один раз).

### Замечание

Существуют три способа записи арифметических выражений: инфиксная (привычная для нас — знак математической операции помещается между operandами), префиксная (знак математической операции помещается перед operandами) и постфиксная (знак математической операции помещается после operandов). Постфиксную запись арифметического выражения называют обратной польской записью.

Рассмотрим алгоритм формирования обратной польской записи математического выражения. Для его реализации нам потребуется два списка: очередь (основной список) и стек (вспомогательный список). Напомним, что математические операции умножения и деления имеют высший приоритет по отношению к сложению и вычитанию. При формировании обратной польской записи будем использовать следующие правила:

- 1) если текущая лексема — число или переменная, то она помещается в очередь;
- 2) если текущая лексема — открывающаяся скобка, то она помещается в стек;
- 3) если текущая лексема — математическая операция и стек пуст или вершиной стека является открывающаяся скобка, то лексема помещается в стек;
- 4) если текущая лексема — математическая операция и стек не пуст и вершиной стека не является открывающаяся скобка, то:
  - а) если вершиной стека является математическая операция одного приоритета с текущей лексемой, то эта операция извлекается из стека и помещается в очередь, а текущая лексема записывается в стек;
  - б) если вершиной стека является математическая операция с приоритетом выше текущей лексемы, то все операции до открывающейся скобки извле-

каются из стека и записываются в очередь, а текущая операция помещается в стек,

в) если вершиной стека является математическая операция с приоритетом ниже текущей лексемы, то текущая лексема помещается в стек;

5) если текущая лексема — закрывающая скобка, то из стека извлекаются все операции до открывающейся скобки и помещаются в очередь, открывающаяся скобка также извлекается из стека;

6) если лексемы закончились и стек оказался не пуст, то все операции извлекаются из стека и помещаются в очередь.

Проиллюстрируем правила формирования обратнойпольской записи на примере математического выражения

$$3 + (4 * a / 7 - 3.5 / 2.1 * 4) * 10 - a.$$

Текущая лексема:	3	+	(	4	*	a	/	7
Стек:		+	( +	( +	* ( +	* ( +	/ ( +	/ ( +
Очередь:	3	3	3	3 4	3 4	3 4 a	3 4 a *	3 4 a * 7
Текущая лексема:	-		3.5		/		(	
Стек:	- ( +		- ( +		/ - ( +		( / - ( +	
Очередь:	3 4 a * 7 /		3 4 a * 7 / 3.5		3 4 a * 7 / 3.5		3 4 a * 7 / 3.5	
Текущая лексема:	2.1			*			4	
Стек:	( / - ( +			* ( / - ( +			*	
Очередь:	3 4 a * 7 / 3.5 2.1			3 4 a * 7 / 3.5 2.1			( / - ( +	
Текущая лексема:	)			)			3 4 a * 7 / 3.5 2.1 4	
Стек:	/ - ( +			+				
Очередь:	3 4 a * 7 / 3.5 2.1 4 *			3 4 a * 7 / 3.5 2.1 4 * / -				
Текущая лексема:	*			10				
Стек:	* +			* +				
Очередь:	3 4 a * 7 / 3.5 2.1 4 * / -			3 4 a * 7 / 3.5 2.1 4 * / - 10				
Текущая лексема:	-				a			
Стек:	-				-			
Очередь:	3 4 a * 7 / 3.5 2.1 4 * / - 10 * +				3 4 a * 7 / 3.5 2.1 4 * / - 10 * + a			

Больше лексем нет, поэтому итоговая очередь, содержащая польскую запись математического выражения, будет выглядеть следующим образом:

$$3 4 a * 7 / 3.5 2.1 4 * / - 10 * + a -$$

Чтобы подсчитать значение выражения по польской записи, необходимо последовательно применять операцию к двум аргументам, стоящим слева от операции. Для рассматриваемого выражения порядок выполнения операций будет иметь следующий вид:

$$\begin{array}{ccccccccc} 1 & 2 & & 3 & 4 & 5 & 6 & 7 & 8 \\ ((3 (((4a*) 7 /) (3.5 (2.1 4 *)) /) -) 10 *) +)a-) \end{array}$$

Реализуйте рассмотренный алгоритм самостоятельно.

*Замечания.* 1. Разбиение исходного выражения на лексемы можно проводить с помощью стандартных функций для работы со строками, а проверку корректности записи математического выражения можно проводить на этапе формирования польской записи выражения.

2. Если выражение начинается с унарного минуса, например « $-7 + 4/2$ » или « $-a*b$ », то рекомендуется свести унарный минус к бинарному минусу добавлением незначащего нуля следующим образом: « $0 - 7 + 4/2$ » или « $0 - a*b$ ».

---

# Глава 17

## РЕАЛИЗАЦИЯ СПИСКОВ С ПОМОЩЬЮ БИБЛИОТЕКИ СТАНДАРТНЫХ ШАБЛОНОВ

В гл. 12 мы рассматривали класс-контейнер *vector* библиотеки стандартных шаблонов STL языка C++, а также итераторы и алгоритмы для работы с этим классом-контейнером. Теперь мы изучим классы-контейнеры STL, предназначенные для реализации списков.

### 17.1. Класс-контейнер *stack*

Стандартный класс-контейнер *stack* очень похож на класс, реализацию которого мы рассматривали в параграфе 16.2. Для работы с данным классом-контейнером необходимо подключить библиотеку `#include <stack>`.

В классе имеется конструктор по умолчанию, который инициализирует пустой стек. Например:

```
stack<int> s;
```

Основные функции данного класса-контейнера приведены в табл. 17.1.

Таблица 17.1

Функции класса-контейнера *stack*

Функция	Описание
<code>empty()</code>	Возвращает <i>true</i> , если исходный стек пуст, и <i>false</i> в противном случае
<code>size()</code>	Возвращает размер стека
<code>top()</code>	Возвращает указатель на вершину стека
<code>pop()</code>	Удаляет элемент из вершины стека
<code>push(val)</code>	Добавляет элемент в вершину стека

Предположим, что мы задали пустой стек

```
stack<int> s;
```

Тогда после выполнения оператора

```
if (s.empty())
{
```

```

    cout << "Стек пуст" << endl;
}
else
{
    cout << "Стек не пуст" << endl;
}

```

на экране появится сообщение: «Стек пуст».

Положить в стек элементы со значениями от 0 до 10 можно следующим образом:

```

for (int j = 0; j <= 10; j++)
{
    s.push(j);
}

```

Вывести содержимое стека на экран можно следующим образом:

```

while (!s.empty())
{
    cout << s.top() << " ";
    s.pop();
}

```

В результате на экране получим

10 9 8 7 6 5 4 3 2 1 0

## 17.2. Класс-контейнер *queue*

Стандартный класс-контейнер *queue* очень похож на класс, реализацию которого мы рассматривали в параграфе 16.5. Для работы с данным классом-контейнером необходимо подключить библиотеку `#include <queue>`

В классе существует конструктор по умолчанию, который инициализирует пустую очередь. Например, пустая очередь, которая будет состоять из целых чисел, может быть создана следующим образом:

```
queue<int> q;
```

Основные функции данного класса-контейнера приведены в табл. 17.2.

Таблица 17.2

Функции класса-контейнера *queue*

Функция	Описание
<code>empty()</code>	Возвращает <i>true</i> , если исходная очередь пуста, и <i>false</i> в противном случае
<code>size()</code>	Возвращает размер очереди
<code>front()</code>	Возвращает указатель на первый элемент очереди

Функция	Описание
<code>back()</code>	Возвращает указатель на последний элемент очереди
<code>pop()</code>	Удаляет первый элемент очереди
<code>push()</code>	Добавляет элемент в конец очереди

Положим в очередь элементы с значениями от 0 до 10, затем выведем содержимое очереди на экран:

```
queue<int> q;
for (int i = 0; i <= 10; i++)
{
    q.push(i);
}
while (!q.empty())
{
    cout << q.front () << " ";
    q.pop();
}
```

В результате на экране получим:

```
0 1 2 3 4 5 6 7 8 9 10
```

### 17.3. Класс-контейнер *list*

Стандартный класс-контейнер *list* очень похож на класс, реализацию которого мы рассматривали в параграфе 16.9. Для работы с данным классом-контейнером необходимо подключить библиотеку `#include <list>`.

Данный класс-контейнер содержит несколько конструкторов:

- *list()* — конструктор по умолчанию, создает пустой список;
- *list(*num*, *val*)* — создает список из *num* элементов, значение которых равно *val*;
- *list(*anotherList*)* — создает список из тех же элементов, что и список *anotherList*.

#### Замечание

---

Здесь и далее *val* имеет тот же тип, что и элементы списка, *iter* — это итератор, *num* — целое, *anotherList* — объект класса-контейнера *list*.

---

Таким образом, задать пустой список *l* из целых чисел можно следующим образом:

```
list<int> l;
```

Основные функции и алгоритмы для работы с данным классом-контейнером представлены в табл. 17.3.

Функции класса-контейнера *list*

Функция	Описание
<code>empty()</code>	Возвращает <i>true</i> , если исходный список пуст, и <i>false</i> в противном случае
<code>size()</code>	Возвращает размер списка
<code>insert(iter, val)</code>	Вставляет элемент <i>val</i> в список непосредственно перед элементом, на который указывает итератор <i>iter</i>
<code>push_front(val)</code>	Вставляет элемент <i>val</i> в начало списка
<code>push_back(val)</code>	Вставляет элемент <i>val</i> в конец списка
<code>remove(val)</code>	Удаляет из списка все элементы, имеющие значение <i>val</i>
<code>erase(iter)</code>	Удаляет из списка элемент, на который указывает итератор <i>iter</i>
<code>front()</code>	Возвращает значение первого элемента списка
<code>pop_front()</code>	Удаляет первый элемент из списка
<code>begin()</code>	Возвращает итератор на начало списка
<code>end()</code>	Возвращает итератор на конец списка
<code>reverse()</code>	Переворачивает исходный список
<code>merge(anotherList)</code>	Объединяет исходный список со списком <i>anotherList</i>
<code>unique()</code>	Удаляет стоящие рядом элементы с одинаковыми значениями, оставляя только один из них

Проиллюстрируем примером работу некоторых функций и алгоритмов для класса-контейнера *list*.

Предположим, что во входном файле, связанным с потоком *in*, записаны числа

1 1 1 2 2 2 2 3 4 5

а список *l* описан следующим образом:

```
list <int> l;
```

Выполним следующие операторы:

```
if (l.empty())
{
    out << "Список пуст" << endl;
}
else
{
    out << "Список не пуст" << endl;
}
out << "Размер " << l.size() << endl;
```

В выходном файле, связанном с потоком *out*, появится следующая запись:

```
Список пуст
Размер 0
```

Выполним далее чтение данных из файла в список:

```
while (in >> value)
{
    l.push_back(value);
}
in.close();
```

Если после этого мы опять выполним операторы

```
if (l.empty())
{
    out << "Список пуст" << endl;
}
else
{
    out << "Список не пуст" << endl;
}
out << "Размер " << l.size() << endl;
```

в выходной файл добавятся строки:

```
Список не пуст
Размер 10
```

Вывести содержимое списка в файл можно с помощью следующего цикла:

```
// описываем итератор для класса-контейнера List
list<int>::iterator iter;
// с помощью итератора проходим по списку слева направо
for (iter = l.begin(); iter != l.end(); iter++)
{
    // выводим значение, на которое указывает итератор
    out << *iter << " ";
}
out << endl;
```

В выходном файле добавится строка:

```
1 1 1 2 2 2 2 3 4 5
```

Перевернем список и опять выведем его в выходной файл:

```
l.reverse();
for (iter = l.begin(); iter != l.end(); iter++)
{
    out << *iter << " ";
}
out << endl;
```

Получим

5 4 3 2 2 2 1 1 1

И, наконец, удалим из списка рядом стоящие повторяющиеся элементы:

```
l.unique();
for (iter = l.begin(); iter != l.end(); iter++)
{
    out << *iter << " ";
}
out << endl;
```

Получим

5 4 3 2 1

#### Замечание

Подумать, что будет выведено в выходной поток, если во входном будет содержаться последовательность 1 1 1 2 2 1 2 2 3 4 5 3.

## 17.4. Решение практических задач с использованием библиотеки STL

1. Данна последовательность натуральных чисел. Перед каждым элементом, равным  $x$ , вставить элемент, равный  $y$ . Входная последовательность целых чисел и заданные числа  $x$  и  $y$  хранятся в файле *input.txt* (в первой строке файла через пробел записаны  $x$  и  $y$ , во второй — последовательность чисел), выходная последовательность целых чисел записывается в файл *output.txt*.

```
#include <fstream>
#include <stack>
using namespace std;
int main()
{
    stack<int> t, t1;
    int i, x, y;
    ifstream in("input.txt");
    ofstream out("output.txt");
    in >> x;
    in >> y;
    while (in >> i) // считываем данные из файла в стек
    {
        t.push(i);
    }
    in.close();
    while (!t.empty()) // пока первый стек не пуст
```

```

{
    // запоминаем значение верхнего элемента из первого стека
    i = t.top();
    t1.push(i); // помещаем это значение во второй стек
    if (i == x) // если это значение равно x, то
    {
        t1.push(y); // во второй стек записываем значение у
    }
    t.pop(); // удаляем верхний элемент из первого стека
}
// выводим содержимое второго стека в выходной файл
while (!t1.empty())
{
    out << t1.top() << " ";
    t1.pop();
}
out.close();
return 0;
}

```

Результат работы программы:

**input.txt**

```

7 0
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

```

**output.txt**

```

0 7 0 7 1 3 0 7 5 2 5 0 7 2 0 7 9 3 0 7 0 7

```

2. Дан файл *input.txt*, компонентами которого являются натуральные числа. Создать на основе файла очередь. Все элементы, равные *x*, заменить на *y*. Результат вывести в выходной файл *output.txt*.

```

#include <iostream>
#include <queue>
using namespace std;
int main()
{
    queue<int> t, t1;
    int i, x, y;
    ifstream in("input.txt");
    ofstream out("output.txt");
    in >> x;
    in >> y;
    while (in >> i) // считываем данные из файла в очередь
    {
        t.push(i);
    }
    in.close();
    while (!t.empty()) // пока первая очередь не пуста
    {
        i = t.front(); // запоминаем значение первого элемента
        if (i != x) // если это значение равно x,
        {
            t1.push(i); // то помещаем это значение во вторую очередь
        }
    }
}

```

```

    }
    else
    {
        t1.push(y); // иначе во вторую очередь помещаем значение у
    }
    t.pop(); // удаляем первый элемент из очереди
}
while (!t1.empty()) // выводим вторую очередь в выходной файл
{
    out << t1.front() << " ";
    t1.pop();
}
out.close();
return 0;
}

```

Результат работы программы:

**input.txt**

```

7 0
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

```

**output.txt**

```

0 0 1 3 0 5 2 5 0 2 0 9 3 0 0

```

3. Дан файл, компонентами которого являются целые числа. Создать на основе данных файла упорядоченную по убыванию структуру данных без использования какого-либо алгоритма сортировки.

```

#include <list>
#include <fstream>
using namespace std;
int main()
{
    list<int> l;
    int value;
    ifstream in("input.txt");
    ofstream out("output.txt");
    // считываем первый элемент из входного потока
    // и помещаем его в список
    in >> value;
    l.push_back(value);
    list<int>::iterator iter; // описываем оператор
    unsigned int i;
    while (in >> value) // пока не достигнут конец входного потока
    {
        // пропускаем все элементы списка, большие value
        for (i = 1, iter = l.begin();
             ((i < l.size()) && (*iter > value));
             i++, iter++);
        // если дошли до конца списка и все числа больше value
        if (i == l.size() && *iter > value)
        {
            l.push_back(value); // то вставляем value в конец списка
        }
    }
}

```

```

    }
else
{
    l.insert(iter, value); // иначе нашли элемент в списке,
} // меньший value, и вставляем value слева от этого элемента
}
in.close();
// выводим содержимое списка в выходной файл
for (iter = l.begin(); iter != l.end(); iter++)
{
    out << *iter << " ";
}
out.close();
return 0;
}

```

Результат работы программы:

*input.txt*

7 7 1 3 7 5 2 5 7 2 7 9 3 7 7

*output.txt*

9 7 7 7 7 7 5 5 3 3 2 2 1

## Упражнения

### Замечание

При решении задачи самостоятельно выберите необходимый класс-контейнер. Свой выбор обоснуйте.

1. Пусть символ # определен в текстовом редакторе как стирающий символ Backspace, т.е. строка *abc#d##c* в действительности является строкой *ac*.

Дан текстовый файл, в котором встречается символ #. Преобразовать его с учетом действия этого символа.

2. В текстовом файле записана без ошибок формула вида:

*<формула>*= *<цифра>*|*M(<формула>, <формула>)*|*m(<формула>, <формула>)*

где | — или; *<цифра>* — 0|1|2|3|4|5|6|7|8|9; *M* обозначает вычисление максимума, *m* — минимума.

Вычислить значение этой формулы. Например, *M(m(3, 5), M(1, 2))* = 3

3. В текстовом файле записана без ошибок формула вида:

*<формула>*= *<цифра>*|*p(<формула>, <формула>)*|*m(<формула>, <формула>)*

где | — или; *<цифра>* — 0|1|2|3|4|5|6|7|8|9; *m(a, b)* = (*a* - *b*) mod 10,

$$p(a, b) = (a + b) \text{ mod } 10.$$

Вычислить значение этой формулы. Например, *m(9, p(p(3, 5), m(3, 8)))* = 6.

4. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все символы, отличные от цифр, а затем все цифры, сохраняя исходный порядок в каждой группе символов.

5. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер заработной платы. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о мужчинах, потом все данные о женщинах, сохраняя исходный порядок в каждой группе сотрудников.

6. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер заработной платы. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о сотрудниках младше 30 лет, потом данные об остальных сотрудниках, отсортировав данные о сотрудниках в каждой группе по размеру заработной платы.

7. Дан файл, содержащий информацию о студентах: фамилия, имя, отчество, номер группы, оценки по трем предметам текущей сессии. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о студентах, обучающихся на 4 и 5, потом данные об остальных студентах, отсортировав данные о студентах в каждой группе по алфавиту.

8\*. Дан многочлен от одной переменной, например:  $y^4 - y + 1$ ,  $z^2 - 8.9z + 1.2z^2$  или  $-5x^3 + 5 - 6.7x^4 - x$ . Разработать класс *Polinom*, поле которого — многочлен, хранящийся в виде связного списка, со следующей функциональностью:

- а) проверка корректности исходной записи многочлена;
- б) представление многочлена в виде списка с полями: коэффициент при переменной, степень;
- в) приведение подобных в многочлене;
- г) распечатка по списку многочлена в нормальном виде (подобные члены в многочлене приведены и расположены в порядке убывания степеней);
- д) подсчет значения многочлена для заданного значения  $x$ ;
- е) умножение многочлена на число;
- ж) сложение двух многочленов;
- з) произведение двух многочленов;
- и) нахождение производной  $n$ -го порядка;
- к) нахождение интеграла  $n$ -го порядка;
- л) нахождение рациональных корней данного многочлена.

Полную структуру класса продумайте самостоятельно. Работу класса продемонстрируйте на примерах.

## **Список литературы**

1. Ахо, А. В. Структуры данных и алгоритмы : пер. с англ. / А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман. — М. : Вильямс, 2016.
2. Программируем на C++ : учеб.пособие : в 3 ч. Ч. 2. Структуры данных и алгоритмы / И. А. Батраева [и др.]. — Саратов : Научная книга, 2004.
3. Батраева, И. А. Программируем на C++ : учеб. пособие : в 3 ч. Ч. 1. Вводный курс / И. А. Батраева [и др.]. — Саратов : Надежда, 2002.
4. Вирт, Н. Алгоритмы и структуры данных : пер. с англ. / Н. Вирт. — СПб. : Невский диалект, 2008.
5. Керниган, Б. У. Язык программирования С : пер. с англ. / Б. У. Керниган, Д. М. Ритчи. — 3-е изд., испр. — М. : Вильямс, 2016.
6. Климова, Л. М. Си++. Практическое программирование. Решение типовых задач / Л. М. Климова. — М. : КУДИЦ-ОБРАЗ, 2001.
7. Круз, Р. Л. Структуры данных и проектирование программ / Р. Л. Круз. — М. : Бином. Лаборатория знаний, 2014.
8. Кубенский, А. А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ / А. А. Кубенский. — СПб. : БХВ-Петербург, 2004.
9. Лаптев, В. В. С++. Объектно-ориентированное программирование / В. В. Лаптев. — СПб. : Питер, 2008.
10. Лафоре, Р. Объектно-ориентированное программирование в C++ : пер. с англ. / Р. Лафоре. — 4-е изд. — СПб. : Питер, 2007.
11. Липпман, С. Б. Язык программирования C++. Вводный курс / С. Б. Липпман, Ж. Лажойе. — 5-е изд. — М. : Вильямс, 2017.
12. Новожилов, О. П. Информатика : учебник для прикладного бакалавриата / О. П. Новожилов. — 3-е изд., перераб. и доп. — М. : Издательство Юрайт, 2016.
13. Огнева, М. В. Основы программирования на языке C++ : в 2 ч. / М. В. Огнева, Е. В. Кудрина. — Саратов : Научная книга, 2008.
14. Огнева, М. В. Структуры данных и алгоритмы: программирование на языке C++. В 2 ч. Ч. 1 / М. В. Огнева, Е. В. Кудрина. — Саратов : Наука, 2013.
15. Павловская, Т. А. С/C++. Программирование на языке высокого уровня / Т. А. Павловская. — СПб. : Питер, 2009.
16. Страуструп, Б. Язык программирования C++ : пер. с англ. / Б. Страуструп — спец. изд. — М. : Бином. Лаборатория знаний, 2015.

17. Кормен, Т. Х. Алгоритмы: построение и анализ : пер. с англ. / Т. Х. Кормен [и др.]. — 3-е изд. — М. : Вильямс, 2013.
18. Трофимов, В. В. Информатика : в 2 т. Т. 2 : учебник для академического бакалавриата / В. В. Трофимов ; отв. ред. В. В. Трофимов. — 3-е изд., перераб. и доп. — М. : Издательство Юрайт, 2017.

## Приложение 1

# Работа в среде Microsoft Visual Studio

Для разработки, отладки и тестирования программы на C++ могут использоваться различные среды программирования. В данном приложении мы рассмотрим основные приемы работы в среде Microsoft Visual Studio, которые позволяют вам создавать и запускать консольные приложения.

1. Запустите среду.
2. В среде выполните команду *File → New Project*.
3. В открывшемся окне *New Project* (рис. П.1):
  - в разделе *Project types* выберите тип *Visual C++*;
  - в разделе *Templates* выберите *Empty Project*;
  - в разделе *Name* укажите имя папки, в которую будут сохраняться все файлы проекта (в качестве примера введем имя *z*);
  - в разделе *Location* укажите местоположение папки на диске;
  - проследите, чтобы был снят флаг в разделе *Create directory for solution*;
  - нажмите кнопку *OK*.

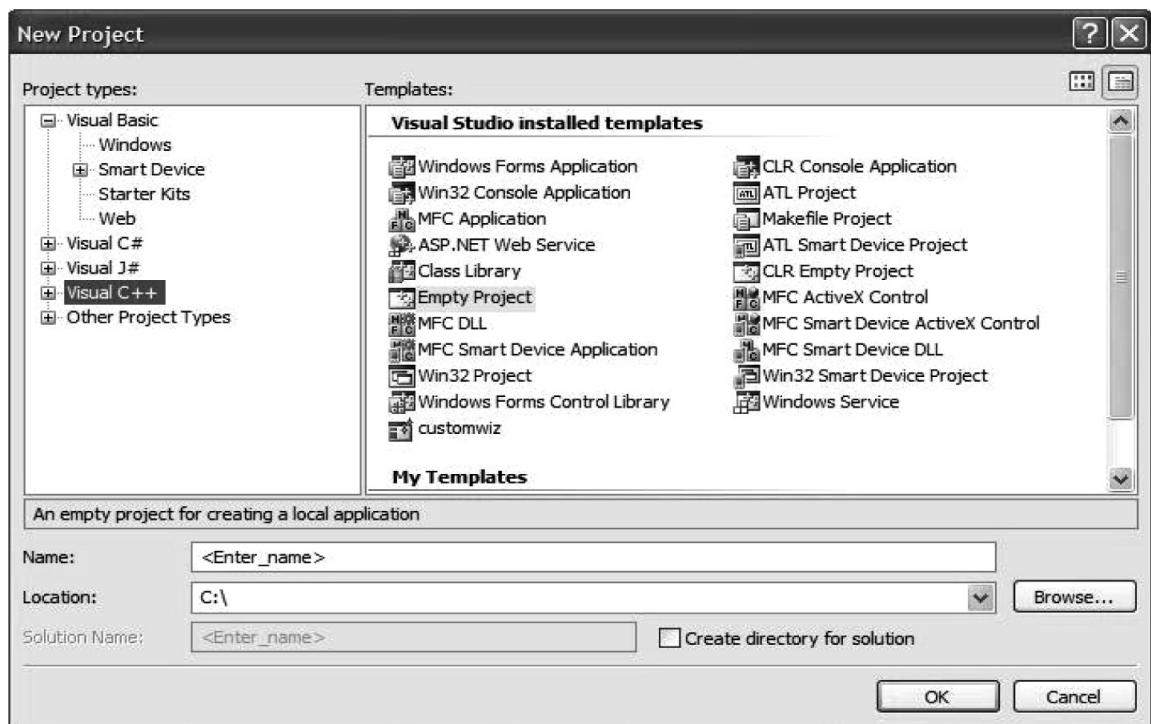


Рис. П.1

4. Откроется окно проекта (рис. П.2).

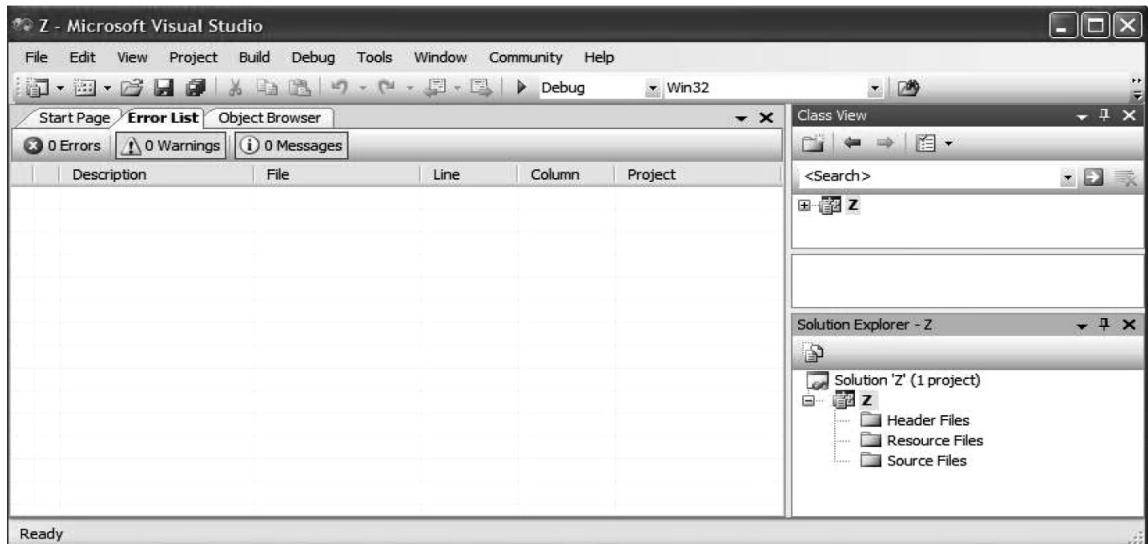


Рис. П.2

5. Во вкладке *Solution Explorer* щелкните правой кнопкой мыши на имени проекта, в нашем случае на **z**.

6. Откроется вспомогательное меню, в котором нужно выполнить последовательность действий *Add* → *New Item*.

7. Откроется диалоговое окно *Add New Item* (рис. П.3):  
— в разделе *Categories* выберите *Code*;  
— в разделе *Templates* выберите *C++File (.cpp)*;  
— в разделе *Name* укажите имя файла (мы введем название **prim**) и нажмите кнопку *Add*.

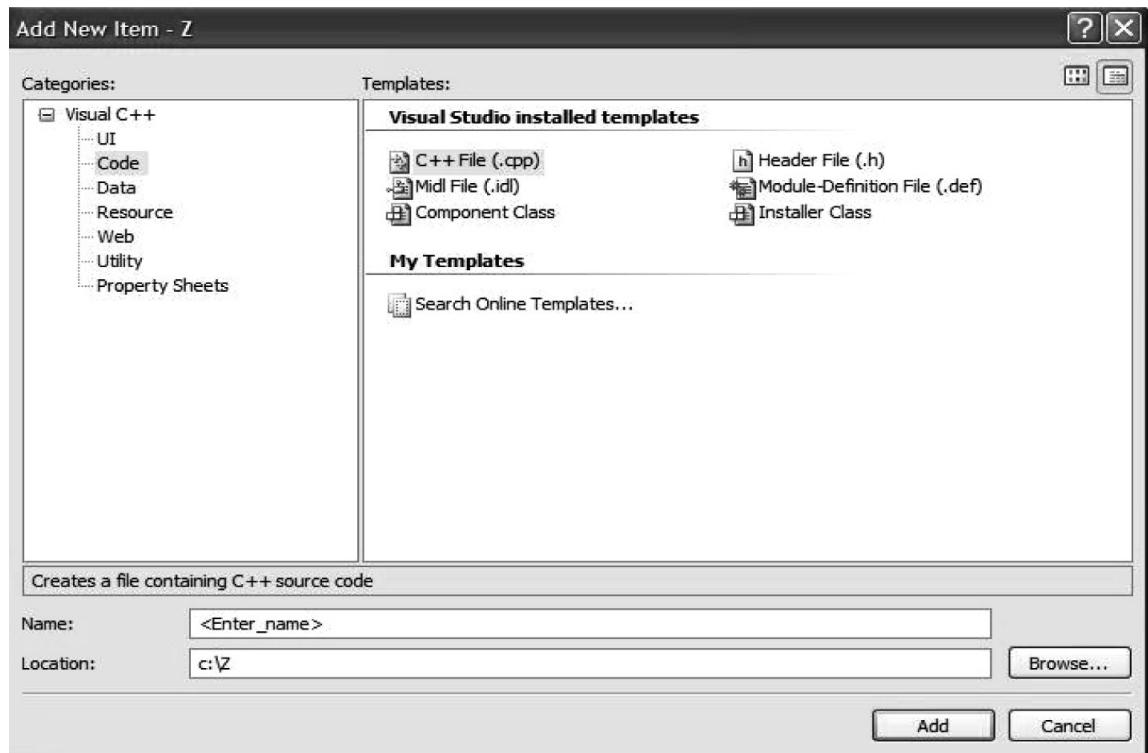


Рис. П.3

8. Откроется окно файла **prim.cpp** (рис. П.4), куда мы и введем код программы, рассмотренный в параграфе 1.2

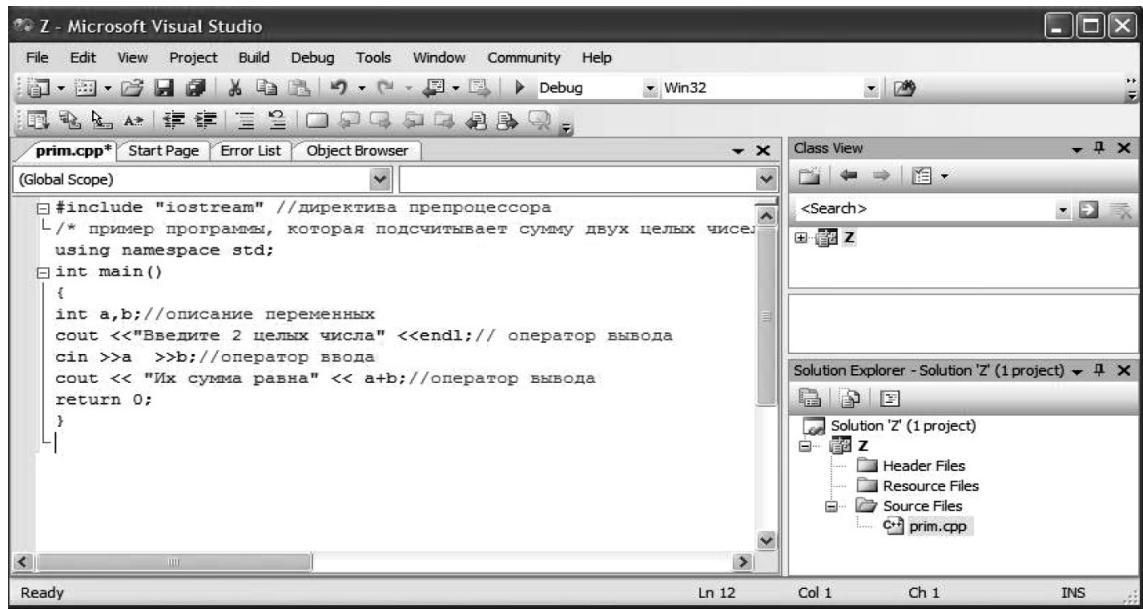


Рис. П.4

Для компиляции кода щелкните правой кнопкой по имени файла во вкладке *Solution Explorer*, в открывшемся вспомогательном меню выберите команду *Compile*.

Для запуска программы из среды Visual Studio нажмите CTRL+F5.

### Замечания

- на Visual C++ директива препроцессора помещается в кавычках;
- если во время компиляции будут обнаружены ошибки, то информация о них будет отображена на вкладке *Error List*;
- после компиляции в папке проекта создастся папка **Debug**, в которой будет находиться exe-файл проекта, готовый к использованию;
- самостоятельно изучите другие возможности Visual Studio с помощью справочной системы.

## Приложение 2

### Ошибки, возникающие при разработке программ

Отладка является непременным этапом при создании любой программы, так как при ее написании обычно допускаются различные ошибки. Ошибки могут быть трех типов: синтаксические, семантические и логические.

*Синтаксические ошибки* возникают в результате нарушения правил написания команд и выявляются на этапе компиляции. При выявлении ошибки компилятор создает сообщение о ее характере, а курсор указывает место в тексте, где эта ошибка обнаружена. В Visual Studio ошибки выводятся на отдельной вкладке *Error List*. Наиболее часто встречающиеся синтаксические ошибки приведены в следующей таблице.

Сообщение об ошибке	Причины возникновения
<i>undeclared identifier</i> (необъявленный идентификатор)	Использование необъявленного идентификатора. Не указан тип идентификатора. Ошибочное написание служебного слова
<i>redefinition</i> (повторное описание)	Повторное описание одного и того же идентификатора (переменной, константы, функции)
<i>syntax error: missing ... before ...</i> (синтаксическая ошибка: пропущен ... перед ... )	Пропущен апостроф при формировании символьной константы. Пропущены кавычки при формировании строковой константы. Пропущена точка с запятой в конце оператора и т.д.
<i>cannot convert from ... to ...</i> (невозможно выполнить неявное преобразование из типа ... в тип ...)	Несовместимы типы переменной и выражения в операторе присваивания. Несовместимы типы фактического и формального параметров при обращении к функции. Тип выражения несовместим с типом индекса при индексировании массива. Несовместимы типы operandов в выражении
<i>end of file found before the left brace ...</i> (конец файла должен находиться слева от позиции ...)	Возможно, в программе несбалансированы фигурные скобки

## Окончание таблицы

Сообщение об ошибке	Причины возникновения
<i>...: illegal, left/right operand has type ...</i> (для операции ... недопустим тип ... левого/правого операнда)	Данная операция не может быть применена к операндам указанного типа, например 10.7 % 2

### Замечание

Более подробно со списком ошибок можно ознакомиться в справочной системе и технической документации по C++.

*Семантические ошибки* возникают в результате использования недопустимых значений параметров или выполнения недопустимых действий над параметрами. Например, при выходе за границу массива, введении недопустимых значений. Выявляются эти ошибки на этапе выполнения программы, о чем также выдается сообщение, например: **Z.exe - обнаружена ошибка. Приложение будет закрыто. Приносим извинения за неудобства.** После этого сообщения программа завершает свою работу.

*Логические ошибки* возникают в связи с ошибочной реализацией алгоритма. Самые простые примеры логических ошибок: вместо операции «+» указали операцию «-», вместо операции сравнения «==» указали операцию присваивания «=». Эти ошибки не приводят к прерыванию программы, но при этом выдаются неверные результаты. Отладка таких ошибок зависит от качества тестовых примеров, составленных вами.

Мы рекомендуем к каждой задаче составить набор тестов, в которых указаны входные значения и определены выходные значения, просчитанные аналитически (без участия программы). Количество тестовых примеров может быть различным в зависимости от сложности программы. Чем сложнее задача, тем больше разрабатывается тестов.

## Приложение 3

### Операции языка C++

Операции приведены в порядке убывания приоритетов. Операции с разными приоритетами разделены чертой.

Операция	Краткое описание
<i>Унарные операции</i>	
::	доступ к области видимости
.	выбор
->	выбор
[ ]	индексация
( )	вызов функции
<тип>( )	конструирование
++	постфиксный инкремент
--	постфиксный декремент
typeid	идентификация типа
dynamic_cast	преобразование типа с проверкой на этапе выполнения
static_cast	преобразование типа с проверкой на этапе компиляции
reinterpret_cast	преобразование типа без проверки
const_cast	константное преобразование типа
sizeof	размер объекта или типа
++	префиксный инкремент
--	префиксный декремент
~	поразрядное отрицание
!	логическое отрицание
+	унарный плюс
-	унарный минус
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти

Окончание таблицы

Операция	Краткое описание
(<тип>)	преобразование типа
.*	выбор
->*	выбор
<i>Бинарные и тернарные операции</i>	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево или извлечение из потока
>>	сдвиг вправо или помещение в поток
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядное И
^	поразрядное исключающее ИЛИ
	поразрядное ИЛИ
&&	логическое И
	логическое ИЛИ
? :	условная операция (тернарная)
=	присваивание
*=	присваивание с умножением
/=	присваивание с делением
%=	остаток от деления с присваиванием
+=	присваивание со сложением
-=	присваивание с вычитанием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием
=	поразрядное ИЛИ с присваиванием
throw	исключение
,	последовательное вычисление

## Приложение 4

### Математические функции

C++ содержит большое количество встроенных математических функций, описание которых хранится в заголовочном файле *math* или *cmath* (в зависимости от версии компилятора). Рассмотрим краткое описание некоторых математических функций. Особое внимание следует обратить на типы операндов и результатов, так как каждая функция имеет несколько перегруженных версий.

Название	Описание
<code>abs(x)</code>	Модуль аргумента, $x$ — целое
<code>acos(x)</code>	Арккосинус (угол в радианах)
<code>asin(x)</code>	Арксинус (угол в радианах)
<code>atan(x)</code>	Арктангенс (угол в радианах)
<code>atan(x,y)</code>	Арктангенс отношения параметра $x/y$ в радианах
<code>ceil(x)</code>	Ближайшее большее целое, $>= x$
<code>cos(x)</code>	Косинус (угол в радианах)
<code>exp(x)</code>	Экспонента $e^x$
<code>fabs(x)</code>	Модуль аргумента, $x$ — вещественное
<code>floor(x)</code>	Ближайшее меньшее целое, $<= x$
<code>log(x)</code>	Логарифм натуральный
<code>log10(x)</code>	Логарифм десятичный
<code>pow (x, y)</code>	Значение $x$ в степени $y$
<code>sin(x)</code>	Синус (угол в радианах)
<code>sqrt(x)</code>	Корень квадратный аргумента
<code>tan(x)</code>	Тангенс (угол в радианах)